PoolTogether micro contest #1 Findings & Analysis Report

2021-09-15

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 July 28—July 31.

Wardens

12 Wardens contributed reports to the PoolTogether micro contest #1 code contest:

1. 0xRajeev
2. gpersoon
3. hickuphh3
4. cmichel
5. pauliax
6. GalloDaSballo
7. shw
8. jonah1005
9. tensors
10. hrkrshnn
11. Jmukesh
12. maplesyrup (heiho1 and thisguy__)

This contest was judged by LSDan.

Final report assembled by moneylegobatman and ninek.

Summary

The C4 analysis yielded an aggregated total of 22 unique vulnerabilities. All of the issues presented here are linked back to their original finding

Of these vulnerabilities, 4 received a risk rating in the category of HIGH severity, 4 received a risk rating in the category of MEDIUM severity, and 14 received a risk rating in the category of LOW severity.

C4 analysis also identified 6 non-critical recommendations and 11 gas optimizations.

Scope

The code under review can be found within the C4 PoolTogether micro contest #1 repository is comprised of 2 smart contracts written in the Solidity programming language and includes ~275 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

[H-01] `onlyOwnerOrAssetManager` can swap Yield Source in `SwappableYieldSource` at any time, immediately rugging all funds from old yield source

Submitted by GalloDaSballo, also found by 0xRajeev and gpersoon

The function `swapYieldSource` SwappableYieldSource.sol L307

Can be called by the owner (deployer / initializer) or Asset Manager. The function will take all funds from the old Yield Source, and transfer them to the new Yield source. Any contract that implement the function `function depositToken() external returns (address)` will pass the check

However, if either the owner or the `assetManager` have malicious intent, this function allows them to instantly rug all funds

1. Create a contract that implements the `function depositToken() external returns (address)`
2. Be the Owner or `AssetManager`
3. Call `setYieldSource` while pointing at your malicious contract
4. Profit

I highly recommend checking that the `YieldSource` is from a trusted registry before allowing this swap.

Alternatively forcing each `Owner` to be a `TimeLock` with at least 48 hours may provide enough security to allow this to be used in practice

PierrickGT (PoolTogether) disputed:

> This is why we will use a multi sig owned by governance to deploy swappable yield sources and manage them. This way, we will avoid these kind of scenarios.

0xean (Judge) commented:

> Agree with warden on the risk here. Will both the AssetManager and the Owner be owned by your governance?

> The YieldSource could easily extract user funds or send them back to the SwappableYieldSource contract and then remove them from there.

PierrickGT (PoolTogether) commented:

> We have removed the `AssetManager` role and `Owner` will be owned by governance who will vet any change of yield source before going through a vote.

[H-02] `redeemToken` can fail for certain tokens

Submitted by cmichel, also found by hickuphh3, pauliax and jonah1005XXX

The `SwappableYieldSource.redeemToken` function transfers tokens from the contract back to the sender, however, it uses the `ERC20.transferFrom(address(this), msg.sender, redeemableBalance)` function for this. Some deposit token implementations might fail as `transferFrom` checks if the contract approved itself for the `redeemableBalance` instead of skipping the allowance check in case the sender is the `from` address.

This can make the transaction revert and the deposited funds will be unrecoverable for the user.

It's recommended to use `_depositToken.safeTransfer(msg.sender, redeemableBalance)` instead.

PierrickGT (PoolTogether) commented:

> Duplicate of https://github.com/code-423n4/2021-07-pooltogether-findings/issues/25

0xean (Judge) commented:

> re-opening this issue and marking #25 as a duplicate of this issue which clearly articulates the potential severity of unrecoverable user funds.

PierrickGT (PoolTogether) resolved:

> This issue has been fixed and we are now using `safeTransfer`: https://github.com/pooltogether/swappable-yield-source/blob/bf943b3818b81d5f5cb9d8ecc6f13ffecd33a1ff/contracts/SwappableYieldSource.sol#L235

[H-03] `setYieldSource` leads to temporary wrong results

Submitted by gpersoon

The use of `setYieldSource` leaves the contract in a temporary inconsistent state because it changes the underlying yield source, but doesn't (yet) transfer the underlying balances, while the shares stay the same.

The function `balanceOfToken` will show the wrong results, because it is based on `_sharesToToken`, which uses `yieldSource.balanceOfToken(address(this))`, that isn't updated yet.

More importantly `supplyTokenTo` will give the wrong amount of shares back: First it supplies tokens to the `yieldsource`. Then is calls `_mintShares`, which calls `_tokenToShares`, which calculates the shares, using `yieldSource.balanceOfToken(address(this))` This `yieldSource.balanceOfToken(address(this))` only contains the just supplied tokens, but doesn't include the tokens from the previous `YieldSource`. So the wrong amount of shares is given back to the user; they will be given more shares than appropriate which means they can drain funds later on (once `transferFunds` has been done).

It is possible to make use of this problem in the following way:

- monitor the blockchain until you see `setYieldSource` has been done
- immediately call the function `supplyTokenTo` (which can be called because there is no access control on this function)

```
// https://github.com/pooltogether/swappable-yield-source/blob/m
function setYieldSource(IYieldSource _newYieldSource) external `
  _setYieldSource(_newYieldSource);
```

```
function _setYieldSource(IYieldSource _newYieldSource) internal
..
    yieldSource = _newYieldSource;

 function supplyTokenTo(uint256 amount, address to) external ove
   ..
    yieldSource.supplyTokenTo(amount, address(this));
    _mintShares(amount, to);
  }

 function _mintShares(uint256 mintAmount, address to) internal {
    uint256 shares = `_tokenToShares`(mintAmount);
    require(shares > 0, "SwappableYieldSource/shares-gt-zero");
    _mint(to, shares);
  }

 function _tokenToShares(uint256 tokens) internal returns (uint2
    uint256 shares;
    uint256 _totalSupply = totalSupply();
..
      uint256 exchangeMantissa = FixedPoint.calculateMantissa(_t
      shares = FixedPoint.multiplyUintByMantissa(tokens, exchang


function balanceOfToken(address addr) external override returns
    return _sharesToToken(balanceOf(addr));
  }

 function _sharesToToken(uint256 shares) internal returns (uint2
    uint256 tokens;
    uint256 _totalSupply = totalSupply();
..
      uint256 exchangeMantissa = FixedPoint.calculateMantissa(yi
      tokens = FixedPoint.multiplyUintByMantissa(shares, exchang
```

Reocommend removing the function `setYieldSource` (e.g. only leave `swapYieldSource`) Or temporally disable actions like `supplyTokenTo`, `redeemToken` and balanceOfToken, after `setYieldSource` and until `transferFunds` has been done.

PierrickGT (PoolTogether) confirmed and resolved:

> PR: https://github.com/pooltogether/swappable-yield-source/pull/4 We've mitigated this issue by removing the `transferFunds` and `setYieldSource` external functions and making `swapYieldSource` callable only by the owner that will be a multi sig wallet for governance pools.

[H-04] `SwappableYieldSource` : Missing same deposit token check in `transferFunds()`

Submitted by hickuphh3, also found by 0xRajeev

`transferFunds()` will transfer funds from a specified yield source `_yieldSource` to the current yield source set in the contract `_currentYieldSource`. However, it fails to check that the deposit tokens are the same. If the specified yield source's assets are of a higher valuation, then a malicious owner or asset manager will be able to exploit and pocket the difference.

Assumptions:

- `_yieldSource` has a deposit token of WETH (18 decimals)
- `_currentYieldSource` has a deposit token of DAI (18 decimals)
- 1 WETH > 1 DAI (definitely true, I'd be really sad otherwise)

Attacker does the following:

1. Deposit 100 DAI into the swappable yield source contract
2. Call `transferFunds(_yieldSource, 100 * 1e18)`
    - `_requireDifferentYieldSource()` passes
    - `_transferFunds(_yieldSource, 100 * 1e18)` is called
        - `_yieldSource.redeemToken(_amount);` → This will transfer 100 WETH out of the `_yieldSource` into the contract
        - `uint256 currentBalance = IERC20Upgradeable(_yieldSource.depositToken()).balanceOf(address(this));` → This will equate to ≥ 100 WETH.

    - `require(_amount <= currentBalance, "SwappableYieldSource/transfer-amount-different");` is true since both are `100 * 1e18`
    - `_currentYieldSource.supplyTokenTo(currentBalance, address(this));` → This supplies the transferred 100 DAI from step 1 to the current yield source
  - We now have 100 WETH in the swappable yield source contract
3. Call `transferERC20(WETH, attackerAddress, 100 * 1e18)` to withdraw 100 WETH out of the contract to the attacker's desired address.

`_requireDifferentYieldSource()` should also verify that the yield sources' deposit token addresses are the same.

```
    function _requireDifferentYieldSource(IYieldSource _yieldSource)
        require(address(_yieldSource) != address(yieldSource), "Swap
                    require(_newYieldSource.depositToken() == yieldS
    }
```

PierrickGT (PoolTogether) acknowledged:

> This exploit was indeed possible when we had the `transferFunds` function but now that we have removed it and funds can only be moved by `swapYieldSource()`, this exploit is no longer possible since we check for the same `depositToken` in `_setYieldSource()`.

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

Oxean (Judge) commented:

> Upgrading to 3 considering the potential for loss of funds

Medium Risk Findings (4)

[M-01] Single-step process for critical ownership transfer/renounce is risky

Submitted by OxRajeev

The `SwappableYieldSource` allows owners and asset managers to set/swap/transfer yield sources/funds. As such, the contract ownership plays a critical role in the protocol.

Given that `AssetManager` is derived from `Ownable`, the ownership management of this contract defaults to `Ownable`'s `transferOwnership()` and `renounceOwnership()` methods which are not overridden here. Such critical address transfer/renouncing in one-step is very risky because it is irrecoverable from any mistakes.

Scenario: If an incorrect address, e.g. for which the private key is not known, is used accidentally then it prevents the use of all the `onlyOwner()` functions forever, which includes the changing of various critical addresses and parameters. This use of incorrect address may not even be immediately apparent given that these functions are probably not used immediately. When noticed, due to a failing `onlyOwner()` or `onlyOwnerOrAssetManager()` function call, it will force the redeployment of these contracts and require appropriate changes and notifications for switching from the old to new address. This will diminish trust in the protocol and incur a significant reputational damage.

See similar **High Risk severity finding** from Trail-of-Bits Audit of Hermez.

See similar **Medium Risk severity** finding from Trail-of-Bits Audit of Uniswap V3:

Recommend overriding the inherited methods to null functions and use separate functions for a two-step address change:

1. Approve a new address as a `pendingOwner`
2. A transaction from the `pendingOwner` address claims the pending ownership change.

This mitigates risk because if an incorrect address is used in step (1) then it can be fixed by re-approving the correct address. Only after a correct address is used in step (1) can step (2) happen and complete the address/ownership change.

Also, consider adding a time-delay for such sensitive actions. And at a minimum, use a multisig owner address and not an EOA.

PierrickGT (PoolTogether) disputed:

> This isn't a security issue but an improper use of the `initialize` function. We do check for address zero so at least the risk of deploying the contract with address zero is excluded. Also, these contracts will be deployed by a multi sig owned by governance so the risk of a single human error is almost null.

Oxean (Judge) commented:

> Disagree with sponsor. A two step process would be a safer implementation. A multi-sig does not remove human error or the potential risk here. It may be an acceptable risk to the team, but still worth highlighting with the given severity.

PierrickGT (PoolTogether) acknowledged:

> We have studied this solution and decided to not implement it since it would make it a pretty tedious process to deploy a swappable yield source, especially through the use of our builder which would mean that a user would have to manually `claimOwnership` after deploying a pool. Plus, this contract will be owned by governance so it will be very difficult to transfer it to another owner or renounce ownership.

[M-02] Use of `safeApprove` will always cause `approveMax` to revert

Submitted by 0xRajeev, also found by pauliax, shw, and cmichel

Unlike `SwappableYieldSource` which uses `safeIncreaseAllowance` to increase the allowance to `uint256.max`, `mStableYieldSource` uses OpenZeppelin's `safeApprove()` which has been documented as (1) Deprecated because of approve-like race condition and (2) To be used only for initial setting of allowance `(current allowance == 0)` or resetting to 0 because it reverts otherwise.

The usage here is intended to allow increase of allowance when it falls low similar to the documented usage in `SwappableYieldSource`. Using it for that scenario will

not work as expected because it will always revert if current allowance is != 0. The initial allowance is already set as `uint256.max` in constructor. And once it gets reduced, it can never be increased using this function unless it is invoked when allowance is reduced completely to 0. See issue page for referenced code.

Recommend Using logic similar to `SwappableYieldSource` instead of using `safeApprove()`.

PierrickGT (PoolTogether) confirmed:

> This issue has been fixed in the following commit: https://github.com/pooltogether/pooltogether-mstable/pull/3/commits/156a990901e6ddff543897905e3ea3d09c78d817

[M-03] Inconsistent balance when supplying transfer-on-fee or deflationary tokens

Submitted by shw, also found by cmichel

The `supplyTokenTo` function of `SwappableYieldSource` assumes that `amount` of `_depositToken` is transferred to itself after calling the `safeTransferFrom` function (and thus it supplies `amount` of token to the yield source). However, this may not be true if the `_depositToken` is a transfer-on-fee token or a deflationary/rebasing token, causing the received amount to be less than the accounted amount.
SwappableYieldSource.sol L211-L212

Recommend getting the actual received amount by calculating the difference of token balance before and after the transfer. For example, re-writing line 211-212 to:

```
uint256 balanceBefore = _depositToken.balanceOf(address(this));
_depositToken.safeTransferFrom(msg.sender, address(this), amount
uint256 receivedAmount = _depositToken.balanceOf(address(this))
yieldSource.supplyTokenTo(receivedAmount, address(this));
```

PierrickGT (PoolTogether) confirmed:

> PR: https://github.com/pooltogether/swappable-yield-source/pull/9

[M-04] Old yield source still has infinite approval

Submitted by tensors, also found by hickuphh3, cmichel and GalloDaSballo

After swapping a yield source, the old yield source still has infinite approval. Infinite approval has been used in large attacks if the yield source isn't perfectly safe (see furucombo).

Recommend decreasing approval after swapping the yield source.

PierrickGT (PoolTogether) confirmed:

> PR: https://github.com/pooltogether/swappable-yield-source/pull/3

Low Risk Findings

[L-01] Initialization function can be front-run with malicious values

Submitted by OxRajeev, also found by cmichel

The `SwappableYieldSource.sol` has a public visibility initialization function that can be front-run, allowing an attacker to incorrectly initialize the contract, if the deployment of this contract does not safely handle initializations via a robust deployment script or a factory contract to prevent front-running.

Impact: Initialization function can be front-run by attackers, allowing them to initialize the contract with malicious values. Also, if initializations are not done atomically with creation, all public/external functions can be accessed before initialization because there are no checks to confirm initializations in those functions.

Reference: See similar High-severity Finding 9 of Trail of Bits audit of Advanced Blockchain and Finding 12 from Trail of Bits audit of Hermez Network.

Recommend ensuring atomic creation+deployment with script or factory contract. Add checks to confirm initialization in public/external functions.

Oxean (Judge) disputed:

> The freeze function is not atomic with the deployment and the script does not enforce that that call is made before moving on to further deployments. The script could enforce that the contract has not been initialized which would at least somewhat mitigate the impacts of a potential front run.

PierrickGT (PoolTogether) commented:

> We are using a factory contract to deploy the Swappable Yield Source so there is no risk of front running: https://github.com/pooltogether/swappable-yield-source/blob/main/deploy/deploy.ts#L92

[L-02] Missing zero-address checks

Submitted by 0xRajeev

Zero-address checks as input validation closest to the function beginning is a best-practice. There are two places where an explicit zero-address check is missing which may lead to a later revert, gas wastage or even token burn.

1. Explicit zero-address check is missing here for `_newYieldSource` and will revert later down the control flow on L256.
2. Missing zero-address check on 'to' address will lead to token burn because imBalances accounts it for the zero-address from which it can never be redeemed using `msg.sender` : `MStableYieldSource.sol` L85

`MStableYieldSource.sol` L94

Recommend adding explicit zero-address checks closest to the function entry.

PierrickGT (PoolTogether) confirmed:

> Swappable Yield Source PR: https://github.com/pooltogether/swappable-yield-source/pull/13

[L-03] `onlyOwner` for `approveMaxAmount()` is risky

Submitted by 0xRajeev

`approveMaxAmount()` is `onlyOwner` while all other privileged functions use `onlyOwnerOrAssetManager`. This modifier should also be `onlyOwnerOrAssetManager` to prevent situations where owner has added asset managers and renounced ownership which will prevent accessing this approval function thereafter.

Recommend change `onlyOwner` to `onlyOwnerOrAssetManager`.

PierrickGT (PoolTogether) acknowledged:

> We have decided to only allow the owner to run `approveMaxAmount` for an added layer of security. For Swappable Yield Sources handled by PoolTogether governance, a multi sig will be used to ensure that not a single person has control of it, this way we limit the risk of the owner renouncing ownership.

0xean (Judge) commented:

> The added security here is dubious given the privileges the asset manager currently has. Would recommend rethinking this approach.

PierrickGT (PoolTogether) commented:

> After discussing with the team, we have decided to make `approveMaxAmount` public in the following commit, since this emergency function should only be called in the case the allowance would have dropped too low.
> https://github.com/pooltogether/swappable-yield-source/pull/9/commits/18e66ae53279d4ef008e271f57fd82500261823f

> Also, we have decided to only allow funds to me moved to the current yield source, so this function shouldn't be used by a malicious actor to steal funds. The changes have been made in the following PR:
> https://github.com/pooltogether/swappable-yield-source/pull/15

[L-04] Overly permissive access control lets anyone approve max amount

Submitted by0xRajeev

Overly permissive access control to lets anyone approve max amount. This may be ok but is inconsistent with `SwappableYieldSource.sol` where the similar function is `onlyOwner`. `onlyOwner`. See issue page for referenced code.

Recommend checking requirements/spec and ensure this is ok or else add `Ownable` inheritance to enforce `onlyOwner` for this function.

PierrickGT (PoolTogether) confirmed patched:

> This issue has been fixed in the following commit:
> https://github.com/pooltogether/pooltogether-mstable/pull/3/commits/41d75dde55fee20caf31b88d3fd61b38caf1663b

[L-05] SwappableYieldSource._requireYieldSource is not a guarantee that you are interacting with a valid yield source

Submitted by GalloDaSballo

`SwappableYieldSource.sol` L74 runs a few checks to see if the function `depositToken` is implemented.

Notice that this is not a guarantee that the target is a valid Yield Source.

This will simply verify that the contract has that method.

Any malicious attacker could implement that function and then set up the Yield Source to steal funds

In order to guarantee that the target is a valid Yield Source, you'd want to create a registry of know Yield Sources, perhaps controlled by governance or by the DAO, and check against that.

Recommend either:

1. Create any contract with just a `function depositToken returns (address)` and you'll be able to add pass the check.

2. Create an on-chain registry of known Yield Sources, either by committee or governance, and use a check against the registry, this will avoid griefing

PierrickGT (PoolTogether) disputed

PierrickGT (PoolTogether) commented:

> Swappable Yield Sources will be deployed by a multi sig owned by governance, `_requireYieldSource` function does indeed simply performs a sanity check to be sure that the yield source address passed is implementing the `depositToken` function. This is to avoid any human error and deploying a swappable yield source that would be unusable cause the address passed wouldn't be a yield source.

> Deployments of a new swappable yield source will be voted by governance, as will a change of yield source, so it would be pretty time and gas consuming to have also to add any new yield source we which to switch to to a registry,

Oxean (Judge) commented:

> Agree with warden that these checks are not sufficient to deter a malicious implementation. Additionally, switching of the yield source looks to be feasible by the owner (presumably the above mentioned multisig) or the AssetManager which is unclear who controls this address. Leaving open.

PierrickGT (PoolTogether) commented:

> We have removed the `AssetManager` role and `Owner` will be owned by governance who will vet any change of yield source before going through a vote.

[L-06] No input validation for while setting up value for immutable state variables

Submitted by JMukesh

Since immutable state variable cant be change after initialization in constructor, their value should be checked before initialization `MStableYieldSource.sol` **L45**

```
constructor(ISavingsContractV2 _savings) ReentrancyGuard() {
```

```
    // @audit --> there should be a input validation

    // As immutable storage variables can not be accessed in the c
    // create in-memory variables that can be used instead.
    IERC20 mAssetMemory = IERC20(_savings.underlying());

    // infinite approve Savings Contract to transfer mAssets from
    mAssetMemory.safeApprove(address(_savings), type(uint256).max)

    // save to immutable storage
    savings = _savings;
    mAsset = mAssetMemory;

    emit Initialized(_savings);
  }
```

Recommend adding a require condition to validate input values.

PierrickGT (PoolTogether) confirmed and patched:

> PR: https://github.com/pooltogether/pooltogether-mstable/pull/4

[L-07] `_requireYieldSource` does not check return value

Submitted by cmichel

The `_requireYieldSource` function performs a low-level status code and parses the return data even if the call failed as it does not check the first return value (`success`). It could be the case that non-zero data is returned even though the call failed, and the function would return `true` .

Check the return value or perform a high-level call using the `_yieldSource` interface.

PierrickGT (PoolTogether) disputed:

> As we noticed while testing the contract, `staticcall` will return the first return value `bool success` at `true` , even if we pass a random wallet address instead of a yield source.

> https://github.com/pooltogether/swappable-yield-source/blob/89cf66a3e3f8df24a082e1cd0a0e80d08953049c/test/SwappableYieldSource.test.ts#L100

> That's why we have decided to check for `depositTokenAddressData.length` and the address returned instead of simply relying on `success`.

> `isValidYieldSource` being initialized at `false` and the fact that we check the return value, I doubt the function would return `true` if non zero data is returned from the staticcall and the call failed.

> https://github.com/pooltogether/swappable-yield-source/blob/653449cfc947894537530007c5388882f418de32a/contracts/SwappableYieldSource.sol#L79

0xean (Judge) commented:

> would recommend following best practices with `staticcall` regardless and still check the boolean return value. Its a trivial amount of gas in a function that wont be called frequently.

PierrickGT (PoolTogether) confirmed and patched:

> This issue has been fixed in the following commit: https://github.com/pooltogether/swappable-yield-source/pull/9/commits/f4cfedc4665dad4635e92a9f96ec9130055dd44d

[L-08] `_requireYieldSource` not always called

Submitted by gpersoon, also found by pauliax

The function initialize of `SwappableYieldSource` checks that the yield source is valid via `_requireYieldSource`. When you change the yield source (via `swapYieldSource` or `setYieldSource`), then the function `_setYieldSource` is called. However `_setYieldSource` doesn't explicitly check the yield source via `_requireYieldSource`.

The risk is low because there is an indirect check, by the following check, which only succeeds is `depositToken` is present in the new yield source:

```
require(_newYieldSource.depositToken() == yieldSource.depos
```

For maintenance purposes it is more logical to always call `_requireYieldSource`, especially if the check would be made more extensive in the future.

```
98
99  function initialize( IYieldSource _yieldSource, uint8 _decim
100     _requireYieldSource(_yieldSource);
101
102  function _requireYieldSource(IYieldSource _yieldSource) int
103     require(address(_yieldSource) != address(0), "SwappableY
104     (, bytes memory depositTokenAddressData) = address(_yiel
105     bool isInvalidYieldSource;
106     if (depositTokenAddressData.length > 0) {
107       (address depositTokenAddress) = abi.decode(depositToke
108       isInvalidYieldSource = depositTokenAddress != address(
109     }
110     require(isInvalidYieldSource, "SwappableYieldSource/inva
111   }
112
113  function _setYieldSource(IYieldSource _newYieldSource) inte
114     _requireDifferentYieldSource(_newYieldSource);
115     require(_newYieldSource.depositToken() == yieldSource.de
116 ...
117
118  function _requireDifferentYieldSource(IYieldSource _yieldSo
119     require(address(_yieldSource) != address(yieldSource), "
120   }
```

Recommend adding the following statement to `_setYieldSource`:

```
_requireYieldSource(_newYieldSource);
```

PierrickGT (PoolTogether) disputed:

> The `_requireYieldSource` function is only used to verify that we setup the swappable yield source with an actual yield source.

> As noted, we already check that `depositToken()` exists in the new yield source, so it would be redundant to also add the `_requireYieldSource` function to perform the same kind of comparaison.

Oxean (Judge) commented:

> Disagree with sponsor. The current implementation doesn't ensure a fully functional yield source is present.

PierrickGT (PoolTogether) commented:

> As previously stated, this contract will be owned by governance who will vet any change of yield source before going through a vote.

[L-09] Variable name or `isInvalidYieldSource` is confusion

Submitted by gpersoon, also found by hickuphh3 and pauliax

The function `_requireYieldSource` of the contract `SwappableYieldSource` has a state variable: `isInvalidYieldSource`

You would expect `isInvalidYieldSource == true` would mean the yield source is invalid However in the source code `isInvalidYieldSource == true` mean the yield source is valid.

This is confusing for readers and future maintainers. Future maintainers could easily make a mistake and thus introduce vulnerabilities.

```
74
75 function _requireYieldSource(IYieldSource _yieldSource) inter
76   require(address(_yieldSource) != address(0), "SwappableYiel
77   (, bytes memory depositTokenAddressData) = address(_yieldSo
78   bool isInvalidYieldSource;
79   if (depositTokenAddressData.length > 0) {
80     (address depositTokenAddress) = abi.decode(depositTokenAd
```

```
81      isInvalidYieldSource = depositTokenAddress != address(0);
82    }
83    require(isInvalidYieldSource, "SwappableYieldSource/invalid
84  }
```

Recommend changing `isInvalidYieldSource` to `isValidYieldSource`

PierrickGT (PoolTogether) confirmed and patched:

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

[L-10] `SwappableYieldSource.sol` : Wrong reporting amount in `FundsTransferred()` event

Submitted by hickuphh3, also found by shw

The `FundsTransferred()` event in `_transferFunds()` will report a smaller amount than expected if `currentBalance > _amount`.

This would affect applications utilizing event logs like subgraphs.

Recommend Updating the event emission to `emit FundsTransferred(_yieldSource, currentBalance);`

PierrickGT (PoolTogether) confirmed and patched:

> This issue has been fixed in the following PR:
> https://github.com/pooltogether/swappable-yield-source/pull/4

[L-11] `SwappableYieldSource` : `setYieldSource()` should check no deposited tokens in current yield source

Submitted by hickuphh3

`setYieldSource()` changes the current yield source to a new yield source. It has similar functionality as `swapYieldSource()`, except that it doesn't transfer deposited funds from the current to the new one. However, it fails to check that it

does not have any remaining deposited funds in the current yield source before the transfer.

It is highly recommended for this check to be in place so that funds aren't forgotten / unintentionally lost.

Recommend adding a require check: `require(yieldSource.balanceOfToken(address(this)); == 0, "SwappableYieldSource/existing-funds-in-current-yield-source")` **before calling `_setYieldSource()

PierrickGT (PoolTogether) acknowledged:

> We have decided to remove `setYieldSource` and `transferFunds` functions. When using `swapYieldSource` to change of yield source, all funds from the old yield source will be moved to the new yield source in one transaction. https://github.com/pooltogether/swappable-yield-source/pull/4

[L-12] Retrieve stuck tokens from `MStableYieldSource`

Submitted by pauliax

Tokens sent directly to the `MStableYieldSource` will be stuck forever. Consider adding a function that allows an admin to retrieve stuck tokens:

- Balance of `mAsset` - total deposited amount of `mAsset`;
- Similar with credit balances as credits are issued as a separate erc20 token.
- All the other tokens.

PierrickGT (PoolTogether) confirmed:

> PR: https://github.com/pooltogether/pooltogether-mstable/pull/8

kamescg (PoolTogether commented:

> LGTM

[L-13] Validation

Submitted by pauliax

Function `supplyTokenTo` should check that `mAssetAmount` and `creditsIssued` > 0 and to != address(0) or if empty to address is provided, it can replace it with msg.sender to prevent potential burn of funds. function `redeemToken` should check that `mAssetAmount` and `creditsBurned` > 0. function `transferERC20` should similarly validate erc20Token, to and amount parameters. function `_mintShares` requires that shares > 0, while `_burnShares` lacks such requirement.

PierrickGT (PoolTogether) disputed:

> This report barely describes which functions or contract should be fixed. This is why I disputed the issue. A proof of concept should have at least been written down and it would have been perfect if recommended mitigation steps were provided in a clear and precise manner.

0xean (Judge) commented:

> Issue is very poorly written. Warden should take time to clearly articulate the issue and impact.

> That being said, I do agree that a check on MStableYieldSource.supplyTokenTo to ensure the `to != address(0)` is reasonable. The mAsset may or may not implement this check, but it would be useful to avoid potential loss of funds.

PierrickGT (PoolTogether) confirmed:

> This issue has been fixed in the following PR:
> https://github.com/pooltogether/pooltogether-mstable/pull/10

[L-14] Some tokens do not have decimals.

Submitted by tensors

There are a few tokens out there that do not use any decimals. As far as I know none of them would be a good yield source, but just in case something comes out, you

may want to include the possibility that decimals = 0. `SwappableYieldSource.sol` L116

Recommend removing the require statement.

PierrickGT (PoolTogether) confirmed:

> PR: https://github.com/pooltogether/swappable-yield-source/pull/2

Non-Critical Findings

- [N-01] Sponsored event not used
- [N-02] Amount should > 0 in `supplyToken()` and `RedeemToken()` in `SwappableYieldSource.sol`
- [N-03] Lack of zero address validation in _requireDifferentYieldSource()
- [N-04] Incorrect comment about memory
- [N-05] `approveMax` in the constructor
- [N-06] Possible enhancements to supply/redeem full balance

Gas Optimizations

- [G-01] SwappableYieldSource.sol: Save depositToken as a storage variable
- [G-02] MStableYieldSource.sol Public functions that should be declared as external to save gas
- [G-03] Redundant zero-address check
- [G-04] MStableYieldSource.sol: `approveMax` can use mAsset instead of savings.underlying()
- [G-05] Adding unchecked directive can save gas
- [G-06] Gas: swapYieldSource
- [G-07] Increase Solc Optimiser Runs
- [G-08] MStableYieldSource.sol: Optimise balanceOf()
- [G-09] SwappableYieldSource.sol: Shorten revert messages
- [G-10] [Optimization] Use 0.8.4 in MStableYieldSource.sol

- [G-11] Use `abi.encodePacked` for gas optimization

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