Fyde Security Review

Pashov Audit Group

Conducted by: Koolex, dirk_y, peanuts, btk

May 27st 2024 - May 30th 2024

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of multiple teams of some of the best smart contract security researchers in the space. Having a combined reported security vulnerabilities count of over 1000, the group strives to create the absolute very best audit journey possible - although 100% security can never be guaranteed, we do guarantee the best efforts of our experienced researchers for your blockchain protocol. Check our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Introduction

A time-boxed security review of the **FydeTreasury/veFyde** and **FydeTreasury/liquid-vault** repositories was done by **Pashov Audit Group**, with a focus on the security aspects of the application's smart contracts implementation.

4. About Fyde

Fyde Liquid Vaults in Fyde allow users to deposit a variety of tokens to gain enhanced performance, yield, and liquidity, while retaining ownership and receiving a liquid vault token, $TRSY, which represents the value of the deposited assets. veFyde manages a whitelist of assets, tracks their weights and emissions, and allows the owner to update and finalize epochs, distributing rewards based on voting balances.

5. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

5.1. Impact

- High - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- Medium - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- Low - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

5.2. Likelihood

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

5.3. Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

6. Security Assessment Summary

review commit hashes:

- 9148db3e070235465257268bcb0bc333c1d4313a
- 6bf1be3dec4b6c9e33929a51bf07539a4ced74f2

fixes review commit hash:

- 5c90a9c06866d1f3572c5463eed919b751aed21a
- 4b169e850b889e82523662a6dcd76cc02a4000ae

Scope

The following smart contracts were in scope of the audit:

in `veFyde`:

- `VoteEscrow`
- `Fyde`
- `FeeDistributor`
- `AssetGuage`

in `veFyde`:

- `RewardsDistributor`
- `YieldManager`
- `YieldToken`
- `YieldTokenFactory`
- `YieldStrategy`
- `sTRSY`
- `OracleModule`
- `TaxModule`

7. Executive Summary

Over the course of the security review, Koolex, dirk_y, peanuts, btk engaged with Fyde to review Fyde. In this period of time a total of **14** issues were uncovered.

Protocol Summary

Protocol Name	Fyde
Repository	https://github.com/FydeTreasury/liquid-vault
Date	May 27st 2024 - May 30th 2024
Protocol Type	Yield management protocol

Findings Count

Severity	**Amount**
High	2
Medium	9
Low	3
Total Findings	**14**

Summary of Findings

ID	Title	Severity	Status
[H-01]	RewardsDistributor calculation doesn't work properly	High	Resolved
[H-02]	Functions cannot be called	High	Resolved
[M-01]	No method to withdraw from EtherFi	Medium	Acknowledged
[M-02]	No slippage check when depositing to Pendle	Medium	Resolved
[M-03]	Low-level call is not checked in swapOn1INCH	Medium	Resolved
[M-04]	depositWETHInEtherfi() is limited to depositing only 1e18	Medium	Resolved
[M-05]	swapOn1INCH() slippage may be ineffective	Medium	Resolved
[M-06]	YieldManager hardcodes slippage to 0	Medium	Acknowledged
[M-07]	Recovering assets in case of emergency could cause an error in accounting	Medium	Acknowledged
[M-08]	Updating boost periods could cause loss of rewards	Medium	Resolved
[M-09]	Loss of rewards due to the boostPeriods design	Medium	Resolved
[L-01]	unsafeTransfer results in cumulativeFees increasing	Low	Resolved
[L-02]	rewardRate will return zero if boostPeriods are set in future timestamp	Low	Resolved

[L-03]	Lack of order validation when setting boost periods	Low	Resolved

8. Findings

8.1. High Findings

[H-01] `RewardsDistributor` calculation doesn't work properly

Severity

Impact: Medium

Likelihood: High

Description

The owner of the `RewardsDistributor` contract can set boost periods for higher reward multipliers by calling `setBoostPeriods`. These periods are used in the `rewardRate` logic when calculating the current reward rate and by association the reward per token.

As suggested in an example by the sponsor, a common configuration for such boost periods is to have the multiplier decrease over time with a series of boost periods. The idea behind this is to start with a higher boost to reward early participants, and tail off the boost multiplier over time.

However, the current `rewardRate` logic doesn't do what's expected. For example, let's assume we have a boost period that starts at timestamp 10 with multiplier 5, and another boost period that starts at timestamp 20 with multiplier 1. Let's also assume that the last update time starts at 0 and the current time is 15.

When the reward rate is calculated now we would have the following in the first iteration of the loop:

```
currTime = 15
nextTimestamp = 15
lastUpdate = 10
timeInPeriod = 15 - 10 = 5
totalTime = 5
rateMultiplier = 0 + 5*5 = 25
```

Since `(lastUpdateTime > boostPeriods[i].timeStamp) == false` we will continue in the for loop to the next boost period:

```
currTime = 15 - 5 = 10
nextTimestamp = 20
lastUpdate = 20
timeInPeriod = 20 - 20 = 0
...
```

This all works as intended and the reward rate is calculated properly. The `lastUpdateTime` variable is now set to 15 (the block timestamp).

Now let's assume the next time someone calls this function the block timestamp is now 25. So we should be in the 1x multiplier period. For the first iteration of the loop:

```
currTime = 25
nextTimestamp = 25
lastUpdate = 15
timeInPeriod = 25 - 15 = 10
totalTime = 10
rateMultiplier = 0 + 10*5 = 50
```

Not only has the time in period been calculated incorrectly (we've only been in the period for 5 seconds, not 10), but `(lastUpdateTime > boostPeriods[i].timeStamp) == true` so we break from the loop.

Now we're rewarding everyone at 5x rather than rewarding at 1x like we should since we've entered the second boost period. This point will apply for all configurations where there are multiple boost periods; we will never make it past the first boost period in the loop once we start the first boost period since we always exit early.

Generally, this function doesn't seem to do what's intended across multiple axes.

Recommendations

The `rewardRate` logic should be updated to properly handle multiple boost periods and generally calculate correctly across period boundaries. Furthermore, I would suggest enforcing the chronological ordering of boost periods in `setBoostPeriods`.

This should also have a suite of associated tests; fuzz testing could be a good option here.

[H-02] Functions cannot be called

Severity

Impact: Medium

Likelihood: High

Description

The `YieldToken` implementation is created within the `YieldTokenFactory` constructor:

```
constructor(address _fyde, address _relayer) {
    TOKEN_IMPLEMENTATION = address(new YieldToken(address
      (this), _fyde, _relayer));
  }
```

The `YieldManager` contract inherits from the `YieldTokenFactory` contract:

```
constructor() YieldTokenFactory(_fyde, _relayer) Ownable
    (msg.sender) ERC1967Proxy(_stratgies, "") {}
```

Therefore, the `YieldManager` will be the owner of the `YieldToken` implementation since it will be the `msg.sender` and thus the `owner` of the implementation:

```
constructor(address _yieldManager, address _fyde, address _relayer) Ownable
    (msg.sender) {
    yieldManager = _yieldManager;
    fyde = _fyde;
    relayer = _relayer;
    baseYieldToken = IYieldToken(address(this));
  }
```

The issue is that the `YieldToken` contract contains two functions that can only be called by the `owner` (i.e., the `YieldManager`):

```
function setYieldManager(address _yieldManager) public onlyOwner {
    yieldManager = _yieldManager;
  }

  function setRelayer(address _relayer) public onlyOwner {
    relayer = _relayer;
  }
```

However, the `YieldManager` cannot call these functions as there is no mechanism provided to do so.

Recommendations

Consider transferring ownership of the `YieldToken` implementation to the `YieldManager.owner()`.

8.2. Medium Findings

[M-01] No method to withdraw from EtherFi

Severity

Impact: Medium

Likelihood: Medium

Description

The `YieldStrategy.sol` contract has the method `depositWETHInEtherfi` that allows the caller to deposit WETH into EtherFi to obtain WEETH.

However, there is currently no method to do the opposite...withdraw the underlying WETH by redeeming the WEETH. As such, the only way to regain the original WETH is to swap via 1INCH, but this depends entirely on the available liquidity in the relevant pools required in the path from WEETH to WETH. I'm not sure why we wouldn't want the `withdraw` equivalent like we have for sDAI and Pendle.

Recommendations

Add a `withdrawWETHInEtherfi` method.

[M-02] No slippage check when depositing to Pendle

Severity

Impact: Low

Likelihood: High

Description

When calling `YieldStrategy.depositToPendle()`, the slippage is set at zero when `router.swapExactTokenForPt()` is called.

```
function depositToPendle(address _underlying, address _market, uint256 _amount)
    external
    returns (uint256)
  {
    (uint256 netPtOut,,) = router.swapExactTokenForPt(
      address(this),
      _market,
>     0,
      defaultApprox(),
      createTokenInputStruct(_underlying, _amount),
      emptyLimit()
    );
    return netPtOut;
  }
```

This is the `swapExactTokenForPt` on the Pendle IPActionSwapPTV3 interface:

```
function swapExactTokenForPt(
        address receiver,
        address market,
        uint256 minPtOut,
        ApproxParams calldata guessPtOut,
        TokenInput calldata input,
        LimitOrderData calldata limit
    ) external payable returns
      (uint256 netPtOut, uint256 netSyFee, uint256 netSyInterm);
```

`minPtOut` is always set to zero.

Recommendations

Set the `minPtOut` as a parameter when calling `depositToPendle()`.

[M-03] Low-level call is not checked in `swapOn1INCH`

Severity

Impact: Low

Likelihood: High

Description

When calling `YieldStrategy.swapOn1INCH()`, the low-level success call is unchecked. The swap may fail without the caller even noticing it. The impact is mitigated by `_minAmountOut`, but if `_minAmountOut` is set to zero, then the unchecked return call value may cause unwarranted issues.

```
function swapOn1INCH(
    address _assetIn,
    uint256 _amountIn,
    address _assetOut,
    uint256 _minAmountOut,
    bytes calldata _swapData
  ) external {
    IERC20(_assetIn).approve(ONEINCH_AGGREGATION_ROUTER, _amountIn);
>   (bool success, bytes memory data) = ONEINCH_AGGREGATION_ROUTER.call
  (_swapData);
    require(IERC20(_assetOut).balanceOf(address
      (this)) >= _minAmountOut, "Slippage Exceeded");
  }
```

Recommendations

Add a check in the swap function.

```
function swapOn1INCH(
    address _assetIn,
    uint256 _amountIn,
    address _assetOut,
    uint256 _minAmountOut,
    bytes calldata _swapData
  ) external {
    IERC20(_assetIn).approve(ONEINCH_AGGREGATION_ROUTER, _amountIn);
    (bool success, bytes memory data) = ONEINCH_AGGREGATION_ROUTER.call
      (_swapData);
+   require(success, "Swap Failed");
    require(IERC20(_assetOut).balanceOf(address
      (this)) >= _minAmountOut, "Slippage Exceeded");
  }
```

[M-04] `depositWETHInEtherfi()` is limited to depositing only 1e18

Severity

Impact: Medium

Likelihood: Medium

Description

The YieldManager owner can deposit WETH into the `eETHliquidityPool` using the `depositWETHInEtherfi()` function:

```
function depositWETHInEtherfi(uint256 _amount) external {
    WETH.withdraw(_amount);
    eETHliquidityPool.deposit{value: 1e18}();
    uint256 eethBal = eETH.balanceOf(address(this));
    eETH.approve(address(weeth), eethBal);
    weeth.wrap(eethBal);
  }
```

- Currently, the function withdraws ETH from the WETH contract and deposits a fixed value of 1e18 into the liquidity pool. Any remaining ETH is left locked in the contract indefinitely.
  - For instance, if the owner calls `depositWETHInEtherfi()` with an amount of 2e18, the contract will withdraw 2e18 ETH from the WETH contract (since 1 WETH = 1 ETH) but only deposit 1e18 into the liquidity pool.
  - The remaining 1e18 ETH will be stuck in the contract.

Recommendations

To resolve this issue, update the `depositWETHInEtherfi()` function to deposit the entire specified amount `_amount` into the liquidity pool:

```
function depositWETHInEtherfi(uint256 _amount) external {
    WETH.withdraw(_amount);
    eETHliquidityPool.deposit{value: _amount}();
    uint256 eethBal = eETH.balanceOf(address(this));
    eETH.approve(address(weeth), eethBal);
    weeth.wrap(eethBal);
  }
```

[M-05] `swapOn1INCH()` slippage may be ineffective

Severity

Impact: High

Likelihood: Low

Description

The `swapOn1INCH()` function attempts to check for slippage as follows:

```
function swapOn1INCH(
    address _assetIn,
    uint256 _amountIn,
    address _assetOut,
    uint256 _minAmountOut,
    bytes calldata _swapData
  ) external {
    IERC20(_assetIn).approve(ONEINCH_AGGREGATION_ROUTER, _amountIn);
    (bool success, bytes memory data) = ONEINCH_AGGREGATION_ROUTER.call
      (_swapData);
    require(IERC20(_assetOut).balanceOf(address
      (this)) >= _minAmountOut, "Slippage Exceeded");
  }
```

This slippage check may be ineffective when there are already tokens of `_assetOut` in the contract. Here's a simple example to illustrate the issue:

- The owner wants to swap 1000 token1 for token2.
  - Assume there are already 500 token2 in the contract.
- The owner sets `_minAmountOut` to 995.
- The swap returns 500 token2 for the 1000 token1.
- The slippage check will pass because the total token2 balance (1000) exceeds `_minAmountOut` (995).

Recommendations

To ensure the slippage check is effective, consider updating the `swapOn1INCH()` function as follows:

```
function swapOn1INCH(
    address _assetIn,
    uint256 _amountIn,
    address _assetOut,
    uint256 _minAmountOut,
    bytes calldata _swapData
  ) external {
    uint256 balBefore = IERC20(_assetOut).balanceOf(address(this));
    IERC20(_assetIn).approve(ONEINCH_AGGREGATION_ROUTER, _amountIn);
    (bool success, bytes memory data) = ONEINCH_AGGREGATION_ROUTER.call
      (_swapData);
    uint256 balAfter = IERC20(_assetOut).balanceOf(address(this));
    require((balAfter - balBefore) >= _minAmountOut, "Slippage Exceeded");
  }
```

[M-06] YieldManager hardcodes slippage to 0

Severity

Impact: High

Likelihood: Low

Description

The current implementation of YieldManager functions `moveAssetToYieldManager()`, `moveAssetToLiquidVault()`, and `depositYieldIntoLiquidVault()` hardcodes the slippage control to 0. This means there is no parameter for slippage control, making all function executions susceptible to significant losses. By setting slippage to 0, these functions accept any amount the relayer returns for swaps or deposits, which can lead to unfavorable outcomes.

The same issue exists in the `YieldStrategy.depositToPendle()` function.

Recommendations

Introduce an `amountOutMinimum` parameter for all affected functions to ensure slippage control.

[M-07] Recovering assets in case of emergency could cause an error in accounting

Severity

Impact: High

Likelihood: Low

Description

`recoverAsset` function is used to withdraw any ERC20 sent to the contract mistakenly. However, this function accepts any address. if the admin passes **TRSY** address, they can extract **TRSY** from the contract which is not supposed to happen.

Furthermore, let's say there is an emergency and the admin wants **TRSY**, `cumulativeFees` will hold the wrong value as the amount withdrawn is not deducted from it. Thus, causing an accounting error in the overall protocol.

Recommendations

Update `cumulativeFees` on `recoverAsset` in case it is **TRSY**

[M-08] Updating boost periods could cause loss of rewards

Severity

Impact: High

Likelihood: Low

Description

`claimFydeEmissions` is used to claim users' rewards. To calculate the user rewards, `earned` view function is utilized.

```
/// @notice Withdraws all earned rewards
  function claimFydeEmissions() external {
                .
                .
                .
    uint256 reward = earned
      (msg.sender, accountBalance, rewardPerToken_, rewards[msg.sender]);
```

RewardsDistributor.sol:L133

`earned` function calculate the due rewards based on `_rewardPerToken`

```
_userReward + _userBalance *
        (_rewardPerToken - userRewardPerTokenPaid[_user]) / PRECISION;
```

RewardsDistributor.sol:L171

`_rewardPerToken` is based on boost periods (if set).

```
function rewardPerToken() public view returns (uint256) {
    return totalSupply == 0
      ? rewardPerTokenStored
      : rewardPerTokenStored
        + (PRECISION * (block.timestamp - lastUpdateTime) * rewardRate
          () / totalSupply);
  }
```

RewardsDistributor.sol#L158-L163

In case the admin updates boost periods, users who didn't claim their due rewards might lose them.

Recommendations

The current design of boost period logic should be changed. For example, disable claiming rewards if boost periods are to be changed. Track the old boost periods Independently so that users can still claim their rewards even after the change.

Another suggestion, set a grace period where users can claim their rewards before any update on the boost periods

[M-09] Loss of rewards due to the boostPeriods design

Severity

Impact: High

Likelihood: Low

Description

`rewardRate()` calculated based on boostPeriods as follows:

```
for (uint256 i; i < boostPeriods.length; ++i) {
      currTime = currTime - timeInPeriod;
      uint256 nextTimeStamp = // if boostPeriods[i].timeStamp in the past, take
      // the current time

                  currTime > boostPeriods[i].timeStamp ? currTime : boostPeriods[i].tim
      uint256 lastUpdate =

                  boostPeriods[i].timeStamp > lastUpdateTime ? boostPeriods[i].timeStam
      timeInPeriod = nextTimeStamp - lastUpdate;
      totalTime += timeInPeriod;

      rateMultiplier += timeInPeriod * boostPeriods[i].multiplier;
      if (lastUpdateTime > boostPeriods[i].timeStamp) break;
    }
```

fRewardsDistributor.sol:L178

If the boostPeriod is in the past, it means expired, therefore, take the current time instead. However, if a user didn't claim during the boost periods, the user would lose the boosted rewards and will earn only based on the normal rate (currentTime - lastUpdateTime).

Please note that, even if the user claims rewards during the boost periods, there will be always some loss. This is because the user can't guarantee to claim the rewards exactly right before the **boostPeriods[i].timeStamp** expires. This issue brings unfairness in terms of reward distribution to the users.

Recommendations

Boosted rewards calculation should be tracked independently from normal rewards per user.

8.3. Low Findings

[L-01] `unsafeTransfer` results in cumulativeFees increasing

`Deposit` function in **FeeDistributor**, performs TRSY transfer as follows:

```
function deposit(address from, uint256 amount) external onlyOwner {
    ERC20(TRSY).transferFrom(from, address(this), amount);
    cumulativeFees += amount;
  }
```

FeeDistributor.sol:L81-L84

Looking at the Implementation of TRSY, it uses a standard ERC20 implementation from Solmate.

Thus, transferFrom doesn't revert, it only returns true/false depending on success/failure. However. there is no check for the returned value.

As a result, silent transfer failures are possible, cumulativeFees will increase with no actual amount transferred to the contract.

Check the returned value, revert if false. Otherwise, use `safeTransferFrom` of SafeTransferLib.

Please apply the same fix for other instances in the codebase.

[L-02] `rewardRate` will return zero if `boostPeriods` are set in future timestamp

It is possible that `rewardRate` will always return zero. This is in case the admin sets boostPeriods in future time. Thus, they will be bigger than block.timestamp (the current time) which makes timeInPeriod always zero, causing totalTime to be zero as well, as a result, rewardRate will return zero.

```
uint256 nextTimeStamp = // if boostPeriods[i].timeStamp in the past, take
      // the current time

                  currTime > boostPeriods[i].timeStamp ? currTime : boostPeriods[i].tim
      uint256 lastUpdate =

                  boostPeriods[i].timeStamp > lastUpdateTime ? boostPeriods[i].timeStam
      timeInPeriod = nextTimeStamp - lastUpdate;
      totalTime += timeInPeriod;
```

This is problematic because when the boost periods are off, the normal rate calculation (based on current time-lastUpdateTime) should still work. However, this is not the case.

Check if all boostPeriods are set in the future, and fallback to the normal rate calculation.

[L-03] Lack of order validation when setting boost periods

Lack of order validation when setting boost periods. If the order is set incorrectly, `rewardRate()` might give wrong results because of this condition:

```
if (lastUpdateTime > boostPeriods[i].timeStamp) break;
```

RewardsDistributor.sol:L193