YO Protocol Core Security Review

Auditors

Gerard Persoon, Lead Security Researcher

Blockdev, Security Researcher

Ladboy233, Security Researcher

Ryan (rscodes), Associate Security Researcher

Report prepared by: Lucas Goiriz

May 29, 2025

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

YO (Yield Optimizer) is a DeFi protocol that helps you easily boost crypto earnings without complications by automatically moving funds across the best-performing pools no matter the blockchain, with the objective of always getting the highest risk-adjusted yield.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of YO Protocol Core according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 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

4 Executive Summary

Over the course of 1 days in total, YO.xyz engaged with Spearbit to review the yoprotocol-core protocol. In this period of time a total of **15** issues were found.

Summary

Project Name	YO.xyz
Repository	yoprotocol-core
Commit	f7dd7c59
Type of Project	Vault, Yield
Audit Timeline	May 20th to May 21st

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	0	0	0
Low Risk	9	2	7
Gas Optimizations	0	0	0
Informational	6	4	2
Total	**15**	**6**	**9**

5 Findings

5.1 Low Risk

5.1.1 Function `__Context_init()` isn't called

Severity: Low Risk

Context: yoVault.sol#L14

Description: Library ERC4626Upgradeable imports library ERC20Upgradeable, which imports ContextUpgradeable.

However the init function of `ContextUpgradeable` isn't called. This risk is limited because this function is empty but this could potentially change in the future.

Recommendation: Consider also calling `__Context_init()` from the `initialize()` function.

YO: Fixed in PR 5.

Spearbit: Fix verified.

5.1.2 Inflation griefing attacks are possible

Severity: Low Risk

Context: yoVault.sol#L14, yoVault.sol#L283

Description: Currently the code uses `ERC4626Upgradeable.sol`'s `_decimalsOffset()` to protect against inflation attacks.

```
function _decimalsOffset() internal view virtual returns (uint8) {
    return 0;
}
```

However, it does not override `_decimalsOffset()`, which means it will be using `10**0 = 1` virtual share. Although this alone prevents the Attacker from outright profiting during an inflation attack, leaving it as `10**0` means griefing attacks causing the user to lose funds are possible.

Sequence: Considering the USDC vaults, the Attacker may carry out the following griefing attack.

1. Deposit 1 wei.
2. Now there are 2 shares and 2 assets (due to ERC4626Upgradeable's handling of virtual shares and assets).
3. `Alice` intends to deposit $1000 usdc.
4. Attacker frontruns by donating $2000 to the contract.
5. Now there are 2 shares and 2002 assets, and the share price is $1001.
6. `Alice`'s transaction gets processed and she gets 0 shares.

`Alice` loses her $1000.

Impact:

1. Loss of funds for victim.
2. Share price would be left at a high value and future deposits will end up getting rounded off too.

Recommendation: The best choice is to deposit some assets into the vault when you deploy. (With a reasonable number of starting shares it will become too expensive to pull off an inflation attack).

Alternative solution with other consequences: Consider overriding `_decimalsOffset` to return 3. (which is the standard number used to prevent inflation attacks).

- Cons related to this alternative solution is that it might unnecessarily increase virtual shares.

YO: Acknowledged. This can only affect the very first deposit/user. We're far beyond that already. Also, our deployment scripts does the first deposit through a private RPC.

Spearbit: Acknowledged.

5.1.3 `onUnderlyingBalanceUpdate()` could revert on `totalSupply()` of 0

Severity: Low Risk

Context: yoVault.sol#L215-L232

Description: In `onUnderlyingBalanceUpdate()` in theory `totalSupply()` could be 0. In that situation the code will revert on yoVault.sol#L222. In that siuation if won't be possible to call `_pause()`.

Recommendation: Consider using `_convertToAssets()`, which prevents a division by 0 and also seems more appropriate to use.

YO: Fixed in PR 7.

Spearbit: Fix verified.

5.1.4 Results of `pendingRedeemRequest()` are misleading

Severity: Low Risk

Context: yoVault.sol#L149-L174, yoVault.sol#L180-L192, yoVault.sol#L269-L271, yoVault.sol#L334-L353

Description: In function `requestRedeem()`, if insufficient assets are present, then the array `_pendingRedeem[]` is filled. In the array, the expected amount of `assetsWithFee` received is stored, based on the `_convertToAssets()` at that moment. A user can retrieve this information via `pendingRedeemRequest()`.

However than the fees are not subtracted yet and there is no easy way to calculate the assets after fees. Additionally the fees can change in the future and then a different amount of assets will be received.

Recommendation: Consider doing one of the following:

- In `pendingRedeemRequest()` subtract the fees from the assets and add a comment to indicate that fees can change.
- Also lock in the fees at the moment of `requestRedeem()`, for example by explicitly storing the fees in `_pendingRedeem[]`.

YO: Acknowledged.

Spearbit: Acknowledged.

5.1.5 Tokens may not be going to respective wallets

Severity: Low Risk

Context: yoVault.sol#L149-L174

Description: From `requestRedeem`, we can see that:

```
function requestRedeem(uint256 shares, address receiver, address owner) external whenNotPaused returns
↪  (uint256) {
    require(shares > 0, Errors.SharesAmountZero());
    require(owner == msg.sender, Errors.NotSharesOwner());
    require(balanceOf(owner) >= shares, Errors.InsufficientShares());

    uint256 assetsWithFee = super.previewRedeem(shares);

    // instant redeem if the vault has enough assets
    if (_getAvailableBalance() >= assetsWithFee) {
        _withdraw(owner, receiver, owner, assetsWithFee, shares);
        emit RedeemRequest(receiver, owner, assetsWithFee, shares, true);
        return assetsWithFee;
    }

    emit RedeemRequest(receiver, owner, assetsWithFee, shares, false);
    // transfer the shares to the vault and store the request
    _transfer(owner, address(this), shares);

    totalPendingAssets += assetsWithFee;
    _pendingRedeem[receiver] = PendingRedeem({
        shares: _pendingRedeem[receiver].shares + shares,
        assets: _pendingRedeem[receiver].assets + assetsWithFee
    });

    return REQUEST_ID;
}
```

It can be seen that `owner` is the wallet that holds the vault share tokens. While `receiver` is the wallet that holds the asset tokens. (`fulfillRedeem` will send the asset tokens there). However in the event of a cancellation:

```
function cancelRedeem(address receiver, uint256 shares, uint256 assetsWithFee) external requiresAuth {
    PendingRedeem storage pending = _pendingRedeem[receiver];
    require(pending.shares != 0 && shares <= pending.shares, Errors.InvalidSharesAmount());
    require(pending.assets != 0 && assetsWithFee <= pending.assets, Errors.InvalidAssetsAmount());

    pending.shares -= shares;
    pending.assets -= assetsWithFee;
    totalPendingAssets -= assetsWithFee;

    emit RequestCancelled(receiver, shares, assetsWithFee);
    // transfer the shares back to the owner

    IERC20(address(this)).transfer(receiver, shares);
}
```

We can see that the code sends it to the `receiver` of the `_pendingRedeem`. (and not the owner since its vault share tokens being sent out instead during cancellation). Users who use 2 wallets for segregation of assets might be affected.

- `owner` for vault share tokens.
- `receiver` for asset tokens.

Overall shouldn't be a big deal unless users are using wallets that are not compatible with either tokens.

Recommendation: During cancellation, since it is the vault share token being sent instead of the asset token, consider making the `owner` the endpoint instead.

YO: Acknowledged.

Spearbit: Acknowledged.

5.1.6 Operator can change share-to-asset exchange rate

Severity: Low Risk

Context: yoVault.sol#L180-L210

Description: Operator can call `fulfillRedeem()` with any values of of `shares` and `assetsWithFee` as long as they are less than the max values specified in `_pendingRedeem[receiver]`. So in an extreme case, it can call:

- `shares = 1` and `assetsWithFee = _pendingRedeem[receiver].assets`.
- `shares = _pendingRedeem[receiver].shares` and `assetsWithFee = 1`.

The specified shares are burned and the specified assets are transferred. This creates an imbalance in `totalSupply()` and `totalAssets()` and thus the new deposits or withdraws may get more or less shares depending on the proportion of shares burned and assets transferred in `fulfillRedeem()`.

Proof of Concept: Run `forge test --mt test_imbalanced_redeem -vv`. This test calls `fulfillRedeem()` from the same state with different parameters and shows how many shares another user can mint after it:

```
function test_imbalanced_redeem() public {
    vm.startPrank({ msgSender: users.alice });
    uint256 depositFeeAmount = _feeOnTotal(depositAmount, depositFee);
    uint256 netDepositAmount = depositAmount - depositFeeAmount;

    depositVault.deposit(depositAmount, users.alice);

    moveAssetsFromVault(netDepositAmount);
    updateUnderlyingBalance(netDepositAmount);

    vm.startPrank({ msgSender: users.alice });
    uint256 aliceShares = depositVault.balanceOf(users.alice);

    depositVault.requestRedeem(aliceShares, users.alice, users.alice);

    vm.roll(block.number + 1);
    usdc.transfer(address(depositVault), netDepositAmount);
    updateUnderlyingBalance(0);

    vm.startPrank({ msgSender: users.admin });
    (uint256 pendingAssets, uint256 pendingShares) = depositVault.pendingRedeemRequest(users.alice);

    uint snapshot = vm.snapshotState();
    depositVault.fulfillRedeem(users.alice, pendingShares, pendingAssets);
    (uint256 pendingAssetsAfterr, uint256 pendingSharesAfter) =
    ↪  depositVault.pendingRedeemRequest(users.alice);

    vm.startPrank({msgSender: users.bob});
    depositVault.deposit(depositAmount, users.bob);
    vm.stopPrank();

    console2.log("when all pending shares and assets are redeemed");
    console2.log("pendingAssetsAfterr", pendingAssetsAfterr);
    console2.log("pendingSharesAfter", pendingSharesAfter);
    console2.log("bob balance", depositVault.balanceOf(users.bob));
    console2.log("\n");

    vm.revertToState(snapshot);
    vm.startPrank({ msgSender: users.admin });
    depositVault.fulfillRedeem(users.alice, 1, pendingAssets);
    (pendingAssetsAfterr, pendingSharesAfter) = depositVault.pendingRedeemRequest(users.alice);

    vm.startPrank({msgSender: users.bob});
    depositVault.deposit(depositAmount, users.bob);
```

```
    vm.stopPrank();
    console2.log("when 1 pending share and all assets are redeemed");
    console2.log("pendingAssetsAfterr", pendingAssetsAfterr);
    console2.log("pendingSharesAfter", pendingSharesAfter);
    console2.log("bob balance", depositVault.balanceOf(users.bob));
}
```

Recommendation: Revert `fulfillRedeem()` if `shares` and `assetsWithFee` aren't in the correct proportion. If trust is placed in the operator to always set the values correctly, then the security of the operator is critical.

YO: Acknowledged. The operator has no economic incentive to fulfill the redeem order with an incorrect amount of assets.

Spearbit: Acknowledged.

5.1.7 `deposit` might coincide with `updateDepositFee`

Severity: Low Risk

Context: yoVault.sol#L252-L256

Description:

```
function updateDepositFee(uint256 newFee) external requiresAuth {
    require(newFee < MAX_FEE, Errors.InvalidFee());
    emit DepositFeeUpdated(feeOnDeposit, newFee);
    feeOnDeposit = newFee;
}
```

Suppose the admin decides to update the deposit fee. Their transaction to update the deposit fee might happen around the same time as an user calling `deposit`. This could possibly cause the user to get slightly less shares than they thought they would get.

- As they might have expected the `feeOnDeposit` fee amount that they saw when they submitted the transaction.

Note: same logic applies to `feeOnWithdraw`.

Recommendation: Consider letting the user specify a maximum acceptable fee during deposit.

- That way, in this rare senario the transaction can revert if the fee amount is beyond something the user wishes to accept.

YO: Acknowledged. This is expected behaviour. In a future version we will have a Router contract that has "minAmountOut" similar to DEXs.

Spearbit: Acknowledged.

5.1.8 Depositors can exploit incorrect price ratio from yield gain to get more shares

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description:

```
function requestRedeem(uint256 shares, address receiver, address owner) external whenNotPaused returns
↪  (uint256) {
    // ...
    // ...

    totalPendingAssets += assetsWithFee;
    _pendingRedeem[receiver] = PendingRedeem({
        shares: _pendingRedeem[receiver].shares + shares,
        assets: _pendingRedeem[receiver].assets + assetsWithFee
    });

    return REQUEST_ID;
}
```

We can see that the shares and assets are stored in the `_pendingRedeem` array, but they are also counted in `totalAssets()` and `totalShares()`. The protocol's intention is to not give yield to shares that are pending for redemptions. However, any amounts of yield coming in between a request and a fulfil will cause a incorrect price ratio that can be exploited.

1. Suppose there are 100 shares and $100 in the vault.
2. `Alice` request a redemption of 50 shares, and the request gets added to pending.
3. $40 yield is added to the vault.
4. Since this senario is the one where the protocol does not give user the yield, the real ratio can be visualised as 50 shares with $90 ($50 left + $40).
5. However, any users depositing during this time will be subjected to the 100 shares and $140 ratio.

Users depositing now in between the transaction will then get more shares than supposed. As when they deposit the price is calculated as `$140 / 100 = $1.40` instead of the real price `$90 / 50 = $1.80`.

Proof of Concept: The following proof of concept shows that with a deposit of `50,000 ETH` you make a profit of `0.01 ETH`.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.29;
import "hardhat/console.sol";
import {Math} from "@openzeppelin/contracts/utils/math/Math.sol";

contract inflate {
    using Math for uint256;
    uint totalSupply = 2409_670558412588463691;
    uint totalAssets = 2525_864718795604928384;
    uint rrAssets     = 100_000000000000000000;
    uint oracleAdd =   2525_864718795604928384 * uint(0.01369863e18) / 100e18;
    uint deposit =    50000_000000000000000000; // 50000 ETH
    function _convertToShares(uint256 assets, uint ts, uint ta,Math.Rounding rounding) internal pure
    ↪  returns (uint256) {
        return assets.mulDiv(ts  + 1 , ta + 1, rounding);
    }
    function _convertToAssets(uint256 shares, uint ts, uint ta,Math.Rounding rounding) internal pure
    ↪  returns (uint256) {
        return shares.mulDiv(ta  + 1, ts  + 1 , rounding);
    }
    constructor() {
       uint rrShares = _convertToShares(rrAssets,totalSupply,totalAssets,Math.Rounding.Floor);
       uint a1;uint a2;
    {   // situation 1
       uint ta1 = totalAssets;
       uint ts1 = totalSupply;
       // add the oracle update
```

```
        ta1  += oracleAdd;
        uint s1 = _convertToShares(deposit,ts1,ta1,Math.Rounding.Floor);
        ts1 += s1;
        ta1 += deposit;
        console.log("include rr: deposit of %d * 1e16 -> %d * 1e16 shares ",deposit /1e16,s1/1e16);
        // now do the rr
        ts1 -= rrShares;
        ta1 -= rrAssets;
        a1 = _convertToAssets(s1,ts1,ta1,Math.Rounding.Floor);
        console.log("Value of deposit = %d * 1e16",a1 / 1e16);
    }
    {  // situation 2 - virtually remove the Request Redeem amounts

        uint ta2 = totalAssets;
        uint ts2 = totalSupply;
        // first do the rr
        ts2 -= rrShares;
        ta2 -= rrAssets;
        // add the oracle update
        ta2 += oracleAdd;
        uint s2 = _convertToShares(deposit,ts2,ta2,Math.Rounding.Floor);
        ts2 += s2;
        ta2 += deposit;
        console.log("exclude rr: deposit of %d * 1e16 -> %d * 1e16 shares",deposit/1e16,s2/1e16);
        a2 = _convertToAssets(s2,ts2,ta2,Math.Rounding.Floor);
        console.log("Value of deposit = %d * 1e16",a2 / 1e16);
    }
        console.log("difference = %d * 1e16",(a1-a2) / 1e16); // difference = 1 * 1e16
    }
}
```

Impact: Depositors make use of temporary incorrect price ratio to mint **more shares**, successfully getting more assets.

Recommendation: Consider updating total shares and total assets to reflect the shares and assets waiting in pending. This is relevant because the reserved `assets` and `shares` in `_pendingRedeem[]` are essentially locked in a predetermined ratio, so they shouldn't be used in any of the other calculations.

Note: its also important in `cancelRedeem()` to release `assets` and `shares` in a synchonized manner as not to disturb the balance.

YO: Acknowledged. A large amount of funds need to put at risk to extract very negligible amount. Adding a withdrawal fee, would completely remove this possibility.

Spearbit: Acknowledged.

5.1.9 `onUnderlyingBalanceUpdate` can be sandwiched

Severity: Low Risk

Context: yoVault.sol#L215-L232

Description: The admin can call `onUnderlyingBalanceUpdate` to increase the `aggregatedUnderlyingBalances` and the updated underlying balances is reflected in the `totalAssets()`.

1. User detect there is a pending `onUnderlyingBalanceUpdate` transaction.
2. The user mint / deposit before the `aggregatedUnderlyingBalances` update.
3. `onUnderlyingBalanceUpdate` transaction get executed.
4. The user `redeem` after the `aggregatedUnderlyingBalances` update to sandwich the increased balance.

Recommendation: Use private RPC such as flashbot to avoid sandwiching.

YO: Acknowledged. We are aware of that and we're using private RPC.

Spearbit: Acknowledged.

5.2 Informational

5.2.1 Consider override `max check` functions to return 0 when contract is paused

Severity: Informational

Context: yoVault.sol#L30

Description: The `yoVault.sol` inherits from ERC4626:

```
function maxDeposit(address) public view virtual returns (uint256) {
    return type(uint256).max;
}

/// @inheritdoc IERC4626
function maxMint(address) public view virtual returns (uint256) {
    return type(uint256).max;
}

/// @inheritdoc IERC4626
function maxWithdraw(address owner) public view virtual returns (uint256) {
    return _convertToAssets(balanceOf(owner), Math.Rounding.Floor);
}

/// @inheritdoc IERC4626
function maxRedeem(address owner) public view virtual returns (uint256) {
    return balanceOf(owner);
}
```

But when the smart contract is paused, no user can deposit and mint and redeem, then `maxDeposit` / `maxMint` / `maxWithdraw` / `maxRedeem` / `maxRedeem` should return 0.

Recommendation: Override `maxDeposit` / `maxMint` / `maxWithdraw` / `maxRedeem` / `maxRedeem` to return 0 when smart contract is paused.

YO: Fixed in PR 4.

Spearbit: Fix verified.

5.2.2 Consider use `block.timestamp` instead of use `block.number` to track balance update

Severity: Informational

Context: yoVault.sol#L215-L232

Description: The code use `block.number` to ensure that the `onUnderlyingBalanceUpdate` update balance in a time-ordered manner. The `block.number` should reflects the current block number on chain, however, in L2 network such as arbitrum (see the Arbitrum vs Ethereum section of the Arbitrum docs):

- The `block.number` reflects l1 block number instead of l2 block number.

Recommendation: Consider use `block.timestamp` instead of `block.number`.

YO: Acknowledged.

Spearbit: Acknowledged.

5.2.3 Two ways to transfer shares

Severity: Informational

Context: yoVault.sol#L149, yoVault.sol#L165, yoVault.sol#L198, yoVault.sol#L209

Description: Function `requestRedeem()` uses `_transfer()` to transfer shares, while function `cancelRedeem()` uses `transfer()`. This is inconsistent and makes the code more difficult to read.

Recommendation: Consider using the same approach in both places.

YO: Fixed in PR 6.

Spearbit: Fix verified.

5.2.4 Incorrect comment for variable `feeOnDeposit`

Severity: Informational

Context: yoVault.sol#L56-L59

Description: The comment for variable `feeOnDeposit` incorrectly mentions `redeemed`.

Recommendation: Consider changing the comment to:

```
- /// @dev the fee charged for the deposits, it's a percentage of the assets redeemed
+ /// @dev the fee charged for the deposits, it's a percentage of the assets deposited
```

YO: Fixed in PR 8.

Spearbit: Fix verified.

5.2.5 Return value of `requestRedeem()` not optimal

Severity: Informational

Context: yoVault.sol#L149-L174

Description: Function `requestRedeem()` returns a `requestId` if the assets can't be redeemed immediately. However this variable also returns `assetsWithFee` when assets can be redeemed. So if the assets can't be redeemed immediately, the only relevant value for `requestId` is 0, which doesn't seem useful. `requestId` isn't used anywhere else in the code.

Note: this is most likely a trace of the similar protocol ERC-7540.

Additionally `assetsWithFee` isn't a useful value, because it isn't equal to real amount of assets that the `receiver` gets, because the fees are added in.

When the assets can't be redeemed immediately then its relatively difficult to know how many assets will be received in the future. It can be inferred from `pendingRedeemRequest()`, but that requires two seperate calls to `pendingRedeemRequest()`: before and after the call to `requestRedeem()`.

Recommendation: Consider doing one or more of the following:

- Rename `requestId` / `REQUEST_ID` to a more meaningful name;
- Return the amount of assets received and possibly the fees in an additional variable;
- Also consider updating the `emits` `RedeemRequest`, `RequestFulfilled` and `RequestCancelled`, which also include fees;
- When the assets can't be redeemed immediately also consider returning the expected asset as a seperate veriable and possibly the fees too.

YO: Acknowledged.

Spearbit: Acknowledged.

5.2.6 Different patterns to update `_pendingRedeem[]`

Severity: Informational

Context: yoVault.sol#L149-L174, yoVault.sol#L180-L192, yoVault.sol#L198-L210

Description: Function `requestRedeem()` uses this pattern to update `_pendingRedeem[]`:

```
_pendingRedeem[receiver] = PendingRedeem({
    shares: _pendingRedeem[receiver].shares + shares,
    assets: _pendingRedeem[receiver].assets + assetsWithFee
});
```

And the functions `fulfillRedeem()` and `cancelRedeem()` use this pattern:

```
PendingRedeem storage pending = _pendingRedeem[receiver];
// ...
pending.shares -= shares;
pending.assets -= assetsWithFee;
```

The second pattern is easier to understand and even saves some gas.

Recommendation: Consider using the same pattern everywhere.

YO: Fixed in PR 9.

Spearbit: Fix verified.