Exponential YoProtocol Review

January 15, 2025

Prepared for Exponential

Conducted by:

Richie Humphrey (devtooligan)

Kurt Willis (phaze)

About the Exponential YoProtocol Review

Exponential is an investment platform that enables DeFi yield opportunities through a vault system. The YoProtocol provides a vault implementation that follows ERC4626 with asynchronous redemptions and automated yield distribution.

About Offbeat Security

Offbeat Security is a boutique security company providing unique security solutions for complex and novel crypto projects. Our mission is to elevate the blockchain security landscape through invention and collaboration.

Scope

The src folder was reviewed at commit 16d83ac.

The following **6 files** were in scope:

- src/AuthUpgradable.sol
- src/Escrow.sol
- src/RolesAuthority.sol
- src/TimelockController.sol
- src/libraries/Errors.sol
- src/yoVault.sol

After our review, we re-reviewed the fixes as of d142927.

Subsequent to that, the project team added some minor changes which we also reviewed up to the commit hash 46d46e7.

Trust Assumptions

The protocol operates with the following trust assumptions and behaviors:

- The vault operator has significant control and requires high trust from users:
  - Can fulfill/cancel redemptions at their discretion
  - Sets price at redemption request time rather than fulfillment
  - Controls underlying balance updates and fee parameters
  - Has authority to execute arbitrary calls with vault's assets
- Redemption mechanics:
  - Requests are processed approximately once per day
  - Users cannot cancel their own redemption requests
  - Only operators can cancel requests in extreme circumstances (e.g. black swan events)
  - Partial fulfillments should not occur
  - Price is fixed at request time rather than fulfillment time

Summary of Findings

The review reports 5 LOW and 1 INFO severity issues related to fee calculations, balance tracking, and specification compliance. No critical, medium, or high severity vulnerabilities were identified.

Identifier	Title	Severity	Fixed

L-01	Total pending assets calculation excludes fees leading to incorrect available balance	Low	d142927
L-02	Precision loss in fee calculations can lead to dust accumulation	Low	d142927
L-03	previewWithdraw returns values without fees	Low	d142927
L-04	Authorization check does not consider value being sent	Low	Acknowledged
L-05	Aggregate balance update susceptible to sandwich attack	Low	Acknowledged
I-01	Cancelled redemption shares are returned to receiver instead of owner	Info	Acknowledged

Detailed Findings

Low Findings

[L-01] Total pending assets calculation excludes fees leading to incorrect available balance

In the yoVault contract's `requestRedeem()` function, `totalPendingAssets` is updated with the fee-subtracted amount rather than the total amount including fees. This causes `_getAvailableBalance()` to overestimate the available assets since fees are not reserved.

```
function requestRedeem(...) {
    uint256 assetsWithFee = super.previewRedeem(shares);
    uint256 assets = assetsWithFee - _feeOnTotal(assetsWithFee, feeOnWithdraw);

    // ...
    totalPendingAssets += assets;  // Should include fee amount
}

function _getAvailableBalance() internal view returns (uint256) {
    uint256 balance = IERC20(asset()).balanceOf(address(this));
    return balance > totalPendingAssets ? balance - totalPendingAssets : 0;
}
```

Recommendation

Update `totalPendingAssets` with the full amount including fees:

```
function requestRedeem(...) {
    uint256 assetsWithFee = super.previewRedeem(shares);
    uint256 assets = assetsWithFee - _feeOnTotal(assetsWithFee, feeOnWithdraw);

    // ...
-   totalPendingAssets += assets;
+   totalPendingAssets += assetsWithFee;
}
```

This ensures that sufficient assets are reserved for both the withdrawal amount and associated fees.

[L-02] Precision loss in fee calculations can lead to dust accumulation

In the yoVault contract, when processing redemptions, assets and fees are calculated twice - once during request creation and again during withdrawal. This double calculation can lead to dust amounts accumulating due to rounding errors from precision loss. Specifically, in `requestRedeem()`, the contract calculates:

```
uint256 assetsWithFee = super.previewRedeem(shares);
uint256 assets = assetsWithFee - _feeOnTotal(assetsWithFee, feeOnWithdraw);
```

Then later in `_withdraw()`, the fee is calculated again based on the stored `assets` amount:

```
uint256 feeAmount = _feeOnRaw(assets, feeOnWithdraw);
```

Recommendation

Store the `assetsWithFee` amount instead of the assets without fee, and modify `_withdraw()` to handle fee calculations in a single place:

```
struct PendingRedeem {
-   uint256 assets;
+   uint256 assetsWithFee;
    uint256 shares;
}

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

      uint256 assetsWithFee = super.previewRedeem(shares);
```

```
-     uint256 assets = assetsWithFee - _feeOnTotal(assetsWithFee, feeOnWithdraw)

      if (_getAvailableBalance() >= assetsWithFee) {
-         _withdraw(owner, owner, owner, assets, shares);
-         emit RedeemRequest(receiver, owner, assets, shares, true);
-         return assets;
+         _withdraw(owner, owner, owner, assetsWithFee, shares);
+         emit RedeemRequest(receiver, owner, assetsWithFee, shares, true);
+         return assetsWithFee;
      }

-     emit RedeemRequest(receiver, owner, assets, shares, false);
+     emit RedeemRequest(receiver, owner, assetsWithFee, shares, false);
      IERC20(address(this)).transferFrom(owner, address(this), shares);

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

      return REQUEST_ID;
  }

- function fulfillRedeem(address receiver, uint256 shares, uint256 assets) exter
+ function fulfillRedeem(address receiver, uint256 shares, uint256 assetsWithFee)
    PendingRedeem storage pending = _pendingRedeem[receiver];
    require(pending.shares != 0 && shares <= pending.shares, Errors.InvalidShares
-   require(pending.assets != 0 && assets <= pending.assets, Errors.InvalidAssets
+   require(pending.assetsWithFee != 0 && assetsWithFee <= pending.assetsWithFee,

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

-   emit RequestFulfilled(receiver, shares, assets);
+   emit RequestFulfilled(receiver, shares, assetsWithFee);
-   _withdraw(address(this), receiver, address(this), assets, shares);
+   _withdraw(address(this), receiver, address(this), assetsWithFee, shares);
  }

  function _withdraw(
      address caller,
      address receiver,
      address owner,
-     uint256 assets,
+     uint256 assetsWithFee,
      uint256 shares
  ) internal override {
+     uint256 feeAmount = _feeOnTotal(assetsWithFee, feeOnWithdraw);
+     uint256 assets = assetsWithFee - feeAmount;

      super._withdraw(caller, receiver, owner, assets, shares);
```

```
        address recipient = feeRecipient;
        if (feeAmount > 0 && recipient != address(this)) {
            IERC20(asset()).safeTransfer(recipient, feeAmount);
        }
    }
```

This approach ensures consistent fee calculation and prevents dust accumulation from rounding errors.

[L-03] previewWithdraw returns values without fees

The yoVault contract does not override the `previewWithdraw()` function from ERC4626, which means it does not account for withdrawal fees when previewing withdraw operations. This breaks the ERC4626 standard requirement that preview functions must accurately reflect all fee deductions.

Recommendation

Consider either:

1. Override `previewWithdraw()` to account for fees:

```
function previewWithdraw(uint256 assets) public view virtual override returns (u
    uint256 shares = super.previewWithdraw(assets);
    return shares + _feeOnRaw(shares, feeOnWithdraw);
}
```

2. Or follow EIP-7540 and revert since the vault only supports asynchronous withdrawals:

```
function previewWithdraw(uint256) public view virtual override returns (uint256)
    revert Errors.UseRequestRedeem();
}
```

This ensures the preview functions accurately reflect the vault's behavior and fee structure.

[L-04] Authorization check does not consider call value being sent

In the yoVault's `manage()` function, the authorization check only validates that the caller is authorized to call a specific function on a target contract, but does not validate whether they are allowed to send value along with the call. This means that any address with authorization to call any specific function can also send the entire ETH balance of the vault:

```
function manage(address target, bytes calldata data, uint256 value) external req
    bytes4 functionSig = bytes4(data);
    require(
        Authority(authority()).canCall(msg.sender, target, functionSig),
        Errors.TargetMethodNotAuthorized(target, functionSig)
    );

    result = target.functionCallWithValue(data, value);
}
```

Recommendation

When granting function-level authorization, be aware that authorized addresses will have access to utilize the vault's entire ETH balance in their calls. Authorization should therefore only be granted to highly trusted addresses. Consider documenting this behavior clearly in the code and external documentation.

```
/// @notice Allows the vault operator to manage the vault
/// @dev Note: Authorized addresses can send the vault's entire ETH balance alon
/// Authorization should only be granted to highly trusted addresses.
function manage(address target, bytes calldata data, uint256 value) external req
```

[L-05] Aggregate balance update susceptible to sandwich attack

The yoVault contract's mechanism for updating the aggregated underlying balances through `onUnderlyingBalanceUpdate()` is vulnerable to sandwich attacks. An attacker can front-run the balance update with a large deposit and back-run with a redemption to profit from the "jump" in asset valuation.

The yoVault implements a multichain yield aggregation system where the total market value across all chains is updated approximately once per day through the `onUnderlyingBalanceUpdate()` function. This creates discrete jumps in asset valuation that can be exploited through sandwich attacks:

1. An attacker monitors the mempool for `onUnderlyingBalanceUpdate()` transactions
2. When they see a transaction that will increase the aggregated balance:
    - Front-run with a large deposit at the pre-update price
    - Let the balance update complete, increasing the price per share
    - Back-run with an instant redemption using `requestRedeem()` at the new higher price Note: The attacker would not necessarily be able to withdraw everything, but they should at least be able to instantly redeem their initial deposit.

The attack works because:

- There is no cooldown period between deposits and withdrawals
- The price changes occur in discrete jumps rather than smoothly over time
- The attacker can predict the price movement by seeing the calldata
- The vault allows instant withdrawals if sufficient assets are available

Severity Explanation

The impact is medium/high due to the potential capture of the yield increases but this is bounded by several factors:

- The attacker needs significant capital for the front-running deposit
- The profit is capped by the size of the yield "jump" (e.g. if yield increases total assets by 1%, maximum theoretical profit is ~1% of the attack deposit)
- The attack only works on positive yield updates
- The attack competes with fees, gas costs, slippage, and other costs related to the attack.

In addition, this risk may be mitigated from fees as well as from the use of a private mempool like Flashbots. As such, the likelihood of such an attack is assesed at low. This results in an overall severity of Low.

Recommendation

This well-known issue of handling jumps in accumulated rewards is not a trivial problem to solve. Many protocols use algorithms such as Masterchef or Synthetix to distribute a fixed reward pool among holders according to their time-weighted contributions to a pool. Other protocols implement a cool-down period before withdrawals to prevent sandwich attacks.

Informational Finding

[I-01] Cancelled redemption shares are returned to receiver instead of owner

In the yoVault contract, when a redemption request is cancelled via `cancelRedeem()`, the shares are returned to the `receiver` address rather than the original `owner` who initiated the redemption request. This behavior contradicts the code comment which indicates the shares should be returned to the owner:

```
function cancelRedeem(address receiver, uint256 shares, uint256 assets) external
    PendingRedeem storage pending = _pendingRedeem[receiver];
    require(pending.shares != 0 && shares <= pending.shares, Errors.InvalidShares
    require(pending.assets != 0 && assets <= pending.assets, Errors.InvalidAssets

    pending.assets -= assets;
```

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

        emit RequestCancelled(receiver, shares, assets);
        // transfer the shares back to the owner
        IERC20(address(this)).transfer(receiver, shares);
    }
```

When `requestRedeem()` is called, it takes both an `owner` and `receiver` parameter, where:

- `owner` is the original holder of the shares who initiates the redemption
- `receiver` is the intended recipient of the underlying assets after redemption

However, if the redemption is cancelled, the shares are sent to the `receiver` rather than being returned to the original `owner`.

Recommendation

Consider whether returning shares to the owner is necessary behavior. Implementing owner tracking for redemption requests would require significant architectural changes.

At minimum, update the code comments and external documentation to clearly reflect that cancelled redemptions will return shares to the designated receiver rather than the original owner.

Additional Recommendations

Architecture

- Consider implementing a more gradual approach to updating underlying asset balances to prevent MEV opportunities from sudden yield accrual as suggested in M-01
- Consider adding time-delayed mechanics for permissionless redemption cancellations to reduce trust requirements
- Consider implementing vault-specific authority contracts rather than sharing a single contract across all vaults
- Add sanity checks to deployment scripts to prevent first depositor attacks

Centralization Risks

The protocol currently requires significant trust in operators who have extensive control over:

- Redemption fulfillment timing and execution

- Asset price determination
- Protocol parameter updates
- Cross-chain asset management

Consider implementing additional controls and restrictions to reduce centralization risks over time as the protocol matures.

Code Quality

1. **Add Input Validation Sanity Checks**

- Constructor VAULT parameter add zero address check (Escrow.sol#L14)
- updateFeeRecipient() add zero address check (Escrow.sol#L156)
- updateMaxPercentage() add check to ensure the new amount is greater than zero (Escrow.sol#L147)

2. **State Variable Shadowed**
   In requestRedeem(), the owner state variable is shadowed by the parameter `owner`. Use `_owner` instead for clarity. (Escrow.sol#L91)

3. **Use internal helper when initializing**
   initialize() should use updateMaxPercentageChange() instead of directly updating the state variable to ensure the event is emitted and limits are enforced. (Escrow.sol#L43)

4. **Improve Documentation**
   Add comprehensive documentation around trust assumptions and operator privileges.