yAudit Goldilocks ivault4626 Review

Review Resources:

- None beyond the code repositories

Auditors:

- spalen
- fedebianu

Table of Contents

Review Summary

Goldilocks ivault4626

This review covers new code for Goldilocks codebase that introduces a vault system built on top of ERC4626 vaults. The system allows users to split their vault positions into Ownership Tokens (OT) and Yield Tokens (YT), with trading capabilities through Uniswap V3 pools.

The new code of the Goldilocks Repo was reviewed over 2 days. 2 auditors performed the code review between March 20th and March 21st, 2025. The review was limited to the latest commit at the start of the review. This was commit 022263468c7d21afceb44d0ac9ba0ea8a1beedf8.

Scope

The scope of the review consisted of the `Goldivault4626.sol` contract at the specific commit.

After the findings were presented to the Goldilocks team, fixes were made and included in the existing PR or inside new PRs.

This review is a code review to identify potential vulnerabilities in the code. The reviewers did not investigate security practices or operational security and assumed that privileged accounts could be trusted. The reviewers did not evaluate the security of the code relative to a standard or specification. The review may not have identified all potential attack vectors or areas of vulnerability.

yAudit and the auditors make no warranties regarding the security of the code and do not warrant that the code is free from defects. yAudit and the auditors do not represent nor imply to third parties that the code has been audited nor that the code is free from

defects. By deploying or using the code, Goldilocks and users of the contracts agree to use the code at their own risk.

Findings Explanation

Findings are broken down into sections by their respective impact:

- Critical, High, Medium, Low impact
  - These are findings that range from attacks that may cause loss of funds, impact control/ownership of the contracts, or cause any unintended consequences/actions that are outside the scope of the requirements.
- Gas savings
  - Findings that can improve the gas efficiency of the contracts.
- Informational
  - Findings including recommendations and best practices.

Medium Findings

1. Medium - Incorrect logic in `_unstakableYT()` prevents `OT` redemption

In the Goldivault4626 contract, `_unstakableYT()` contains a logical error that incorrectly calculates how many `YT` tokens a user can unstake.

Technical Details

`_unstakableYT()` is called during `redeemOwnership()` and `_redeemOwnership()` operations to determine how many `YT` tokens should be unstaked from a user. However, when a user attempts to unstake more than they have staked (`unstakeAmount > _ytStaked`), the function returns `unstakeAmount - _ytStaked` instead of the expected `_ytStaked` (the maximum amount the user has available to unstake).

This logical error can lead to incorrect unstaking calculations when users attempt to redeem ownership tokens. Subsequently, when the contract attempts to burn the full redemption

amount with `YieldToken(yt).burnYT(msg.sender, amount)`, the transaction will fail because the user won't have enough unstaked `YT` tokens available for burning.

Impact

Medium. Failed redemptions will occur when users attempt to unstake more tokens than they have staked.

Recommendation

Change the logic in the `_unstakableYT()` function to correctly return the maximum amount a user can unstake:

```
/// @notice Calculate the maximum amount of YT the user can unstake
function _unstakableYT(address user, uint256 unstakeAmount) internal view returns
(uint256) {
  return FixedPointMathLib.min(unstakeAmount, ytStaked[user]);
}
```

Developer Response

Fixed here.

2. Medium - Lost value on deposit and withdrawal fees

Technical Details

The flow `buyYT()` can be optimized to minimize pay deposit and withdraw fees to `depositVault`. Vault used in tests has a withdraw fee.

After the swap from `ot` to `depositVault` token, the user redeems vault tokens and pays withdraw fee. Sends the received `depositToken` to the contract which deposits it back to the same `depositVault`. The user can send `depositVault` tokens directly to the contract without paying the fees.

Provided `oriBGT` also has deposit and withdrawal fees, currently set to 0.

Impact

Medium. Buy YT flow loses value by paying fees for unneeded deposits and withdrawals.

Recommendation

Change the lines 206-210 for following:

```
    if (dtNeeded == 0) {
      ERC4626(depositVault).redeem(proceeds, msg.sender, address(this));
    } else {
      uint256 contractShares = ERC4626(depositVault).convertToShares(dtNeeded);
      ERC4626(depositVault).redeem(proceeds - contractShares, msg.sender,
address(this));
    }
```

`convertToShares()` is used to ensure that the withdrawal fee is paid by the user.

Developer Response

Fixed here.

3. Medium - Vault is losing asset in the sell flow

Technical Details

In the sell flow, the vault contract spent shares to get OT tokens. Spent amount of shares is repaid by the user in the form of `depositToken` and deposited back into `depositVault` to get an equal amount of spent shares. `repayAmount` is calculated using `convertToAssets()` function which won't account in deposit fee. This will result in less shares for the vault contract in because it will pay a deposit fee when depositing received user asset.

Provided deposit vault token `oriBGT` also has deposit and withdrawal fees, currently set to 0.

Impact

Medium. The vault will lose value on each sell YT because of paying deposit fees.

Recommendation

Change the way `repayAmount` is calculated to account for deposit fees:

```
- uint256 repayAmount = ERC4626(depositVault).convertToAssets(vaultSpend);
+ uint256 repayAmount = ERC4626(depositVault).previewMint(vaultSpend);
```

Developer Response

Fixed here.

Low Findings

1. Low - Redemption flow doesn't check unstaked `YT` tokens before unstaking

The redemption process is inefficient because it doesn't first check if a user has sufficient unstaked `YT` tokens before attempting to unstake additional tokens. This design flaw wastes gas by potentially unstaking tokens unnecessarily and force the user to stake it again after redemption.

Technical Details

The current redemption flow in Goldivault4626's `redeemOwnership()` and `_redeemOwnership()` directly calculates and unstakes `YT` tokens through `_unstakeYT(unstakableAmount)` without first checking if the user already has enough unstaked `YT` tokens to satisfy the redemption request. This means:

1. Gas is wasted on unnecessary unstaking operations when a user already has sufficient unstaked `YT`
2. Users will have their `YT` tokens unexpectedly unstaked and are forced to stake them again

Impact

Low. The current implementation creates a poor user experience where users must manually manage their staked vs unstaked token balances.

Recommendation

Redesign the redemption flow to optimize user experience and improve readability:

```
function redeemOwnership(uint256 amount) external nonReentrant {
  _redeemOwnership(amount, true);
}

function sellYT (uint256 ytAmount, uint256 dtAmountMin, uint256 amountInMax)
external nonReentrant {
  ...
```

```
    _redeemOwnership(amount, false);
  ...
}

function _redeemOwnership(uint256 amount, bool checkVaultConclusion) internal {
  if(amount == 0) revert InvalidRedemption();

  // check if vault has concluded
  bool shouldBurnYT = true;
  if (checkVaultConclusion) {
    uint256 remainingTime = block.timestamp > endTime ? 0 : endTime -
block.timestamp;
    shouldBurnYT = remainingTime > 0;
  }

  _updateClaimableUnderlying(msg.sender);

  if (shouldBurnYT) {
    uint256 unstakedYT = YieldToken(yt).balanceOf(msg.sender);

    // only unstake what's needed
    if (unstakedYT < amount) {
      uint256 ytToUnstake = amount - unstakedYT;
      if(ytToUnstake > ytStaked[msg.sender]) revert InvalidUnstake();
      _unstakeYT(ytToUnstake);
    }

    YieldToken(yt).burnYT(msg.sender, amount);
  }

  // burn OT tokens and redeem underlying
  OwnershipToken(ot).burnOT(msg.sender, amount);
  ERC4626(depositVault).redeem(ERC4626(depositVault).convertToShares(amount),
msg.sender, address(this));
  depositTokenAmount -= amount;

  emit OwnershipTokenRedemption(msg.sender, amount);
}
```

Developer Response

Acknowledged, we decided to remove unstakableYT here and will make it clear on the UI that YT should be staked before selling or redeeming.

2. Low - Incorrect check for a flash loan amount

Technical Details

In `buyYT()` function, the contract checks if it has enough vault tokens to cover whole flash loan amount. It uses `convertToAssets()` to validated amount but vault used in tests has withdraw fee. Vault implementation doesn't account in the fee in function `convertToAssets()` meaning that the received amount will be lower than expected.

Impact

Low. `buyYT()` function will fail if the whole vault amount is needed for a flash loan.

Recommendation

Use `previewRedeem()` which accounts for withdrawal fees.

```
-
if(ERC4626(depositVault).convertToAssets(ERC20(depositVault).balanceOf(address(th
is))) < dtNeeded) revert FlashLoanFailed();
+
if(ERC4626(depositVault).previewRedeem(ERC20(depositVault).balanceOf(address(this
))) < dtNeeded) revert FlashLoanFailed();
```

Developer Response

Fixed here.

Gas Saving Findings

1. Gas - Unnecessary zero-address check

Technical Details

`_updateClaimableUnderlying()` contains a check for the zero address that consumes gas but provides no actual value, as the function is only called internally within the contract with `msg.sender` as parameter.

Impact

Gas savings.

Recommendation

Remove the check.

Developer Response

Fixed here.

2. Gas - Use `withdraw()` instead of `redeem()` with `convertToShares()`

Technical Details

The `Goldivault4626` contract uses `redeem()` with `convertToShares()` when withdrawing assets from the `ERC4626` vault. This approach is less efficient than using `withdraw()` directly, which handles the conversion internally.

Impact

Gas savings.

Recommendation

Replace instances of `redeem(convertToShares(x))` with direct `withdraw(x)` calls throughout the contract.

Developer Response

Fixed here.

3. Gas - Unnecessary amount check

Technical Details

In function `_unstakeYT()`, check `(amount > ytStaked[msg.sender]` is not needed because the amount passed into the function is always calculated using `_unstakableYT()`.

Impact

Gas savings.

Recommendation

Remove line 335 to save gas.

Developer Response

Acknowledged, we removed the unstakableYT function here and believe this line should remain.

4. Gas - Cast variable only once

Technical Details

In functions `buyYT()` and `sellYT()`, variables `depositToken` and `depositVault` are cast multiple times to ERC20 or ERC4626. Additionally, using local variables will save gas instead of accessing storage variables.

Impact

Gas savings.

Recommendation

Cast variables only once as local variables for gas savings and better readability.

Developer Response

Fixed here.

Update: Removed 2 local variables in buyYT here as they caused stack too deep errors.

Informational Findings

1. Informational - Incomplete interface `IGoldivault4626`

Technical Details

Interface `IGoldivault4626` is incomplete. It defines only the `deposit` and `redeemOwnership` functions but is missing other crucial external functions from `Goldivault4626`.

Impact

Informational.

Recommendation

Provide the correct interface for easier integrations.

Developer Response

Fixed here.

2. Informational - Misleading name `tokenDecimals`

Technical Details

Variable `tokenDecimals` is used to define value of one token, e.g. 1e18. But the name `tokenDecimals` is more appropriate for specifying the number of token decimals, e.g. 18.

In the constructor, parameter `_tokenDecimals` description is "Decimals of underlying asset" which, in our case, would be 18.

Impact

Informational.

Recommendation

Rename the variable `tokenDecimals` to `inputRatioAmount` or something similar.

Developer Response

Fixed here.

3. Informational - Vault losses will cause `_claimableUnderlyingPerYT()` to revert

`Goldivault4626` can work only with up-only `depositVaults`.

Technical Details

The yield calculation in `_claimableUnderlyingPerYT()` assumes that the vault's ratio can only increase or stay the same.

If `newRatio` is less than `oldRatio`, the subtraction will revert due to underflow. This could happen in scenarios such as:

- vault strategy losses
- exploits or hacks of the underlying protocols
- emergency withdrawals from the vault

As `_claimableUnderlyingPerYT()` is called by `stake()` and `unstake()`, this will lock users' funds and prevent new staking until the ratio difference becomes positive again. The issue is temporary and self-resolves when the vault's ratio returns to positive growth.

Impact

Informational.

Recommendation

Add this behaviour to the documentation and add a custom revert to handle this scenario:

```
if (newRatio < oldRatio) revert NegativeYield();
```

If you plan to use negative yield vaults, additional changes are needed to handle it.

Developer Response

Fixed here.

4. Informational - Inappropriate name `underlyingPerTokenDebt`

Technical Details

Variable `underlyingPerTokenDebt` is to store underlying value per PT token. Token debt is not used so the name should be revised.

Impact

Informational.

Recommendation

Rename the variable `underlyingPerTokenDebt` to `underlyingPerYT` and update the description.

Developer Response

I believe it is used as the debt as it is subtracted from the claimable underlying per YT on line 370. I have updated the name to be clear here.

Final remarks

The `Goldivault4626` contract is designed to manage a vault that accepts deposits of an underlying token and splits them into two types of tokens: Ownership Tokens (OT) and Yield Tokens (YT). This mechanism allows users to separate the principal and yield components of their investments. It is not a fork of the popular Pendle protocol but rather a simpler

approach to token splitting, utilizing an external DEX, specifically a Uniswap V3 pool, for trading YT.

Test coverage is minimal, covering only the happy path of deposit and redemption flows. Complex functions, such as buying and selling YT, are not covered by tests. It is recommended to add tests for all functions before deploying the contract in production. Additionally, the tests should deploy external Uniswap V3 pools with OT and the deposit token to verify correct integration. Fuzzing is advised to catch edge cases and potential rounding or accounting issues.