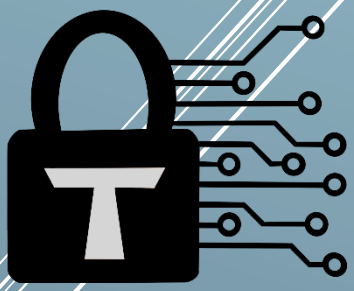
Trust Security

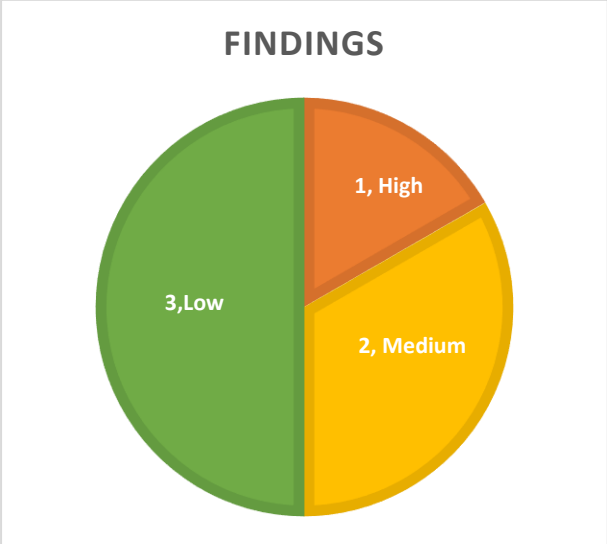


Smart Contract Audit

Reserve Protocol – Staking Contract

16/05/24

Executive summary

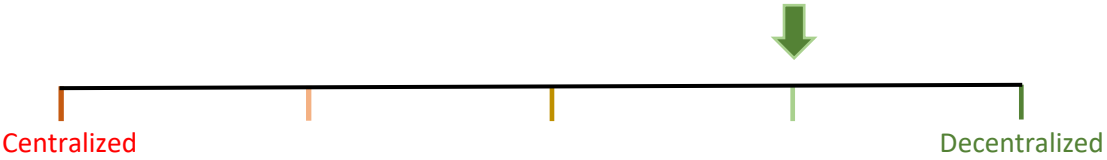


Category	Stablecoin
Auditor	HollaDieWaldfee gjaldon

Findings

Severity	Total	Fixed	Open	Acknowledged
High	1	1	-	-
Medium	2	2	-	-
Low	3	3	-	-

Centralization score



Signature

Digitally signed by Trust
Date: 2024.05.16 12:51:58 +03'00'

Document properties

Versioning

Version	Date	Description
0.1	16/05/2024	Client report

Contact

Trust

trust@trust-security.xyz

Introduction

Trust Security serves as a long-term security partner of the Reserve Protocol. It has conducted the audit at the customer's request. The audit is focused on uncovering security issues and additional bugs contained in the code defined in scope. Additional recommendations have been given when appropriate.

Scope

The following repository is in scope of the audit:

- staking-contract

Repository details

- **Repository URL:** https://github.com/reserve-protocol/staking-contract
- **Commit hash:** 58fa91db137626f1132ee44627b5d4a713165b0f
- **Mitigation hash:** e548e8f0dcb6d5b321f63471f108c96079df950b

The audit has started at **Commit hash** and all changes up to and including **Mitigation hash** have been reviewed. During the audit, an additional file, *ERC4626Router.sol*, was introduced which has also been reviewed.

About Trust Security

Trust Security has been established by top-end blockchain security researcher Trust, in order to provide high quality auditing services. Trust is the leading auditor at competitive auditing service Code4rena, reported several critical issues to Immunefi bug bounty platform and is currently a Code4rena judge.

About the Auditors

HollaDieWaldfee is a renowned security expert with a track record of multiple first places in competitive audits. He is a Lead Senior Watson at Sherlock and Lead Auditor for Trust Security and Renascence Labs.

Gjaldon is a DeFi specialist who enjoys numerical and economic incentives analysis. He transitioned to Web3 after 10+ years working as a Web2 engineer. His first foray into Web3 was achieving first place in a smart contracts hackathon and then later securing a project grant to write a contract for Compound III. He shifted to Web3 security and in 3 months achieved

top 2-5 in two contests with unique High and Medium findings and joined exclusive top-tier auditing firms.

Disclaimer

Smart contracts are an experimental technology with many known and unknown risks. Trust Security assumes no responsibility for any misbehavior, bugs or exploits affecting the audited code or any part of the deployment phase.

Furthermore, it is known to all parties that changes to the audited code, including fixes of issues highlighted in this report, may introduce new issues and require further auditing.

Methodology

In general, the primary methodology used is manual auditing. The entire in-scope code has been deeply looked at and considered from different adversarial perspectives. Any additional dependencies on external code have also been reviewed.

Findings

High severity findings

TRST-H-1 Rounding in GenericMultiRewardsVault leads to loss of rewards

- **Category:** Rounding issues
- **Source:** GenericMultiRewardsVault.sol
- **Status:** Fixed

Description

There exist two calculations in *GenericMultiRewardsVault* which incur a loss due to rounding. In one instance, there can be a complete loss of the reward tokens even under reasonable assumptions.

1) Rounding loss in *_calcRewardsEnd()*

```
return SafeCast.toUint48(block.timestamp + (amount / uint256(rewardsPerSecond)));
```

The rounding loss here is up to **rewardsPerSecond - 1** which should almost certainly be acceptable. The reward paid out in one second should be a dust value for any reasonable usage of the contract. However, this is not guaranteed and if for example **1 ETH - 1 wei** is provided for **amount** and **rewardsPerSecond = 1 ETH**, **1 ETH - 1 wei** is lost.

2) Rounding loss in *_accrueRewards()*

```
// {qRewardTok} = {qRewardTok} * {qShare} / {qShare}
deltaIndex = (accrued * uint256(10 ** decimals())) / supplyTokens;
```

The rounding loss here is up to **1 wei** of **accrued** for each whole supply token (**supplyTokens / 10 ** decimals()**).

It is very reasonable to assume that many supply tokens will be staked. In case of a RToken that tracks USD, one supply token would be worth roughly **1 USD**.

So, a number like **1e6** whole supply tokens is very likely (even **1e7** or **1e8** should be safe). This would be **1m USD** - **100m USD** staked.

Consider a reward token like **WBTC** with **8** decimals. Almost certainly **accrued < 1 WBTC** and all rewards are lost forever (e.g., **(1e7 WBTC * 1e21 / (1e8 whole supply tokens * 1e21) = 0**). Note that the supply token will have **21** decimals.

Recommended mitigation

The recommendation for the rounding loss in *_calcRewardsEnd()* is contained in the diff provided in finding TRST-M-2.

For the rounding loss in *_accrueRewards()*, the following fix is suggested.

```
@@ -158,7 +160,7 @@ contract GenericMultiRewardsVault is ERC4626, Ownable {
```

```
            rewardsEndTimestamp: rewardsEndTimestamp,
            lastUpdatedTimestamp: SafeCast.toUint48(block.timestamp),
            rewardsPerSecond: rewardsPerSecond,
-           index: ONE,
+           index: ONE * 1e18,
            ONE: ONE
        });
        distributorInfo[rewardToken] = distributor;
@@ -325,7 +327,7 @@ contract GenericMultiRewardsVault is ERC4626, Ownable {

        if (supplyTokens != 0) {
            // {qRewardTok} = {qRewardTok} * {qShare} / {qShare}
-           deltaIndex = (accrued * uint256(10 ** decimals())) / supplyTokens;
+           deltaIndex = (accrued * uint256(10 ** decimals()) * 1e18) / supplyTokens;
        }

        // {qRewardTok} += {qRewardTok}
@@ -344,14 +346,14 @@ contract GenericMultiRewardsVault is ERC4626, Ownable {
        // If user hasn't yet accrued rewards, grant them interest from the strategy
beginning if they have a balance
        // Zero balances will have no effect other than syncing to global index
        if (oldIndex == 0) {
-           oldIndex = rewardIndex.ONE;
+           oldIndex = rewardIndex.ONE * 1e18;
        }

        uint256 deltaIndex = rewardIndex.index - oldIndex;

        // Accumulate rewards by multiplying user tokens by rewardsPerToken index and
adding on unclaimed
        // {qRewardTok} = {qShare} * {qRewardTok} / {qShare}
-       uint256 supplierDelta = (balanceOf(_user) * deltaIndex) / uint256(10 **
decimals());
+       uint256 supplierDelta = (balanceOf(_user) * deltaIndex) / uint256(10 **
decimals()) / 1e18;
```

deltaIndex should be expanded by **18** decimals. The precision is then sufficient to prevent a significant rounding loss even in an adverse scenario with low reward token decimals, high reward token value and high supply token supply. At the same time, expansion by **18** decimals poses no risk for overflows to occur.

Team response

Fixed in commits dacc928 and 68bab44.

To prevent scenarios where a bad reward token can DoS the contract, emergency functionality for the **owner** to remove reward tokens and for users to exit the vault has been introduced.

Mitigation review

The recommendations for the rounding issues have been implemented and no issues have been found in the emergency functionality.

Medium severity findings

TRST-M-1 Implementation of GenericMultiRewardsVault totalAssets() is not compliant with ERC4626

- **Category:** Interoperability issues
- **Source:** GenericMultiRewardsVault.sol
- **Status:** Fixed

Description

GenericMultiRewardsVault does not override OZ's *totalAssets()* function, and so calling *totalAssets()* returns the vault's **asset** balance.

However, *GenericMultiRewardsVault* enforces a 1-1 exchange rate between **shares** and **assets**. As a result, if additional **assets** are sent to the contract, they are accounted for in *totalAssets()* but cannot be withdrawn by users.

Due to the incorrect result of *totalAssets()*, ERC4626 compatibility is broken.

Recommended mitigation

Since the contract enforces a 1-1 exchange rate between **shares** and **assets**, a new **totalDeposited** variable can be introduced that is incremented for deposits and decremented for withdrawals. It accurately tracks the accessible **assets** within the vault. *totalAssets()* should be overridden to return this new variable.

Team response

Fixed.

Mitigation review

The recommendation has been implemented.

TRST-M-2 Reward tokens in GenericMultiRewardsVault get permanently lost when totalSupply = 0

- **Category:** Logical flaws
- **Source:** GenericMultiRewardsVault.sol
- **Status:** Fixed

Description

No rewards are accrued in *GenericMultiRewardsVault._accrueRewards()* when **totalSupply() == 0**.

```
    function _accrueRewards(IERC20 _rewardToken, uint256 accrued) internal {
        uint256 supplyTokens = totalSupply();
        uint256 deltaIndex;


        if (supplyTokens != 0) {
            // {qRewardTok} = {qRewardTok} * {qShare} / {qShare}
            deltaIndex = (accrued * uint256(10 ** decimals())) / supplyTokens;
```

```
        }


        // {qRewardTok} += {qRewardTok}
        rewardInfos[_rewardToken].index += deltaIndex;
        rewardInfos[_rewardToken].lastUpdatedTimestamp =
SafeCast.toUint48(block.timestamp);
    }
```

The problem is that **accrued** reward tokens that don't increment the reward token **index**, are stuck in the contract, and cannot be recovered.

Recommended mitigation

Rewards that are not paid out due to **totalSupply() == 0**, should be tracked in a variable so that they can be paid out in a later reward payment.

The proposed solution also keeps track of the tokens lost due to rounding in *GenericMultiRewardsVault._calcRewardsEnd()*, which is a problem discussed in TRST-H-1.

```
@@ -107,6 +107,7 @@ contract GenericMultiRewardsVault is ERC4626, Ownable {
     IERC20[] public rewardTokens;
     mapping(IERC20 rewardToken => RewardInfo rewardInfo) public rewardInfos;
     mapping(IERC20 rewardToken => address distributor) public distributorInfo;
+    mapping(IERC20 rewardToken => uint256 leftoverRewards) public leftoverRewards;

     mapping(address user => mapping(IERC20 rewardToken => uint256 rewardIndex))
public userIndex; // {qRewardTok}
     mapping(address user => mapping(IERC20 rewardToken => uint256 accruedRewards))
public accruedRewards; // {qRewardTok}
@@ -151,7 +152,7 @@ contract GenericMultiRewardsVault is ERC4626, Ownable {
         uint256 ONE = 10 ** rewardTokenDecimals;
         uint48 rewardsEndTimestamp = rewardsPerSecond == 0
             ? SafeCast.toUint48(block.timestamp)
-            : _calcRewardsEnd(0, rewardsPerSecond, amount);
+            : _calcRewardsEnd(0, rewardsPerSecond, amount, rewardToken);

         rewardInfos[rewardToken] = RewardInfo({
             decimals: rewardTokenDecimals,
@@ -212,7 +213,7 @@ contract GenericMultiRewardsVault is ERC4626, Ownable {
         uint256 remainder = prevEndTime <= block.timestamp
             ? 0
             : uint256(rewards.rewardsPerSecond) * (prevEndTime - block.timestamp);
-        uint48 rewardsEndTimestamp =
_calcRewardsEnd(SafeCast.toUint48(block.timestamp), rewardsPerSecond, remainder);
+        uint48 rewardsEndTimestamp =
_calcRewardsEnd(SafeCast.toUint48(block.timestamp), rewardsPerSecond, remainder,
rewardToken);

         rewardInfos[rewardToken].rewardsPerSecond = rewardsPerSecond;
         rewardInfos[rewardToken].rewardsEndTimestamp = rewardsEndTimestamp;
@@ -247,7 +248,7 @@ contract GenericMultiRewardsVault is ERC4626, Ownable {

         uint48 rewardsEndTimestamp = rewards.rewardsEndTimestamp;
         if (rewards.rewardsPerSecond > 0) {
-            rewardsEndTimestamp = _calcRewardsEnd(rewards.rewardsEndTimestamp,
rewards.rewardsPerSecond, amount);
+            rewardsEndTimestamp = _calcRewardsEnd(rewards.rewardsEndTimestamp,
rewards.rewardsPerSecond, amount, rewardToken);
             rewardInfos[rewardToken].rewardsEndTimestamp = rewardsEndTimestamp;
         }

@@ -263,15 +264,19 @@ contract GenericMultiRewardsVault is ERC4626, Ownable {
     function _calcRewardsEnd(
         uint48 rewardsEndTimestamp,
```

```
         uint256 rewardsPerSecond,
-        uint256 amount
-    ) internal view returns (uint48) {
+        uint256 amount,
+        IERC20 rewardToken
+    ) internal returns (uint48) {
+        amount += leftoverRewards[rewardToken];
         if (rewardsEndTimestamp > block.timestamp) {
             // {qRewardTok} += ({qRewardTok/s} * ({s} - {s}))
             amount += rewardsPerSecond * (rewardsEndTimestamp - block.timestamp);
         }

         // {s} = {s} + ({qRewardTok} / {qRewardTok/s})
-        return SafeCast.toUint48(block.timestamp + (amount /
uint256(rewardsPerSecond)));
+        uint48 endTimestamp = SafeCast.toUint48(block.timestamp + (amount /
uint256(rewardsPerSecond)));
+        leftoverRewards[rewardToken] = amount - (endTimestamp - block.timestamp) *
rewardsPerSecond;
+        return endTimestamp;
     }

     function getAllRewardsTokens() external view returns (IERC20[] memory) {
@@ -326,6 +331,8 @@ contract GenericMultiRewardsVault is ERC4626, Ownable {
         if (supplyTokens != 0) {
             // {qRewardTok} = {qRewardTok} * {qShare} / {qShare}
             deltaIndex = (accrued * uint256(10 ** decimals())) / supplyTokens;
+        } else {
+            leftoverRewards[_rewardToken] += accrued;
         }

         // {qRewardTok} += {qRewardTok}
```

Team response

Fixed.

Mitigation review

The recommendation has been implemented.

Low severity findings

TRST-L-1 Exchange rate in GenericStakedAppreciatingVault can be inflated to grief users

- **Category:** Inflation attacks
- **Source:** GenericStakedAppreciatingVault.sol
- **Status:** Fixed

Description

GenericStakedAppreciatingVault implements a virtual share + decimal offset approach to mitigate inflation attacks. While this prevents an attacker from plausibly earning a profit from exchange rate manipulation, a manipulated exchange rate can still cause harm to users.

For example, if an attacker is willing to incur a loss, they can donate **100e18** assets as reward, and users would not receive any shares for depositing up to **~1e16** assets.

The issue is Low severity since an attacker can't earn a profit and the griefing is only possible for a short time until the first deposit is performed.

Recommended mitigation

It is recommended to make an initial deposit in the same transaction that the vault is deployed. An initial deposit of **1e18** assets is a standard amount that prevents exchange rate inflation.

Team response

No change in code, we'll deposit **1e18** units of the staking token immediately after deployment, likely in the same transaction.

Mitigation review

The issue has been addressed by making an initial deposit.

To rule out the issue, the first deposit must be made in the same transaction as the deployment. An attacker can pre-fund the contract and rewards start accruing immediately. Only a deposit in the same transaction as the deployment can raise the assets and shares amounts so that reward accrual cannot manipulate them.

TRST-L-2 GenericMultiRewardsVault.claimRewards() can be front-run to cause reverts

- **Category:** Front-running attacks
- **Source:** GenericMultiRewardsVault.sol
- **Status:** Fixed

Description

GenericMultiRewardsVault.claimRewards() reverts when the **rewardAmount** for a user is zero. Since it is possible to claim rewards for another user, the function can be forced to revert.

In a scenario where the caller is another contract, this unexpected revert can impact the functionality of the contract.

Recommended mitigation

It is recommended to **continue** and to proceed with the next token in the **_rewardTokens** array instead of reverting when **rewardAmount=0**.

```
@@ -91,7 +93,7 @@ contract GenericMultiRewardsVault is ERC4626, Ownable {
             uint256 rewardAmount = accruedRewards[user][_rewardTokens[i]];

             if (rewardAmount == 0) {
-                revert Errors.ZeroRewards(_rewardTokens[i]);
+                continue;
             }

             accruedRewards[user][_rewardTokens[i]] = 0;
```

Team response

Fixed.

Mitigation review

The recommendation has been implemented.

TRST-L-3 Approval can revert in Router's depositChained()

- **Category:** Logical issues
- **Source:** ERC4626Router.sol
- **Status:** Fixed

Description

ERC4626Router uses IERC20's *approve()* in its *depositChained()* function which expects a **bool** to be returned.

```
function depositChained(IERC4626[] memory vaults, uint256 amount) external {
  // ... snip ...
  uint256 depositAmount = asset.balanceOf(address(this));
  asset.approve(address(vault), depositAmount);
  // ... snip ...
}
```

Some tokens, such as **USDT**, do not return a **bool** for their ERC20 functions. The contract will not be able to deposit such assets.

Recommended mitigation

Use *SafeERC20.forceApprove()* which handles *approve()* functions without return values and tokens with approval race condition protections. This makes the *ERC4626Router* work with a much wider range of ERC20 tokens.

Team response

Fixed.

Mitigation review

The recommendation has been implemented.

Additional recommendations

Remove redundant RewardAdded event

It has been determined that the **RewardAdded** event can be removed from the *GenericStakedAppreciatingVault.addRewards()* function since all necessary information is already emitted by the **RewardDistributionUpdated** event in the *GenericStakedAppreciatingVault._updateRewards()* function.

```
         _updateRewards();

-        emit RewardAdded(rewardTracker.rewardAmount, rewardTracker.rewardPeriodStart,
rewardTracker.rewardPeriodEnd);
     }
```

GenericMultiRewardsVault does not need to override decimals() function

Using the implementation of *decimals()* in OZ's *ERC4626* contract works correctly and does not need to perform a staticcall into the **asset** contract each time *decimals()* is called. The recommendation is to simply remove the *decimals()* function from *GenericMultiRewardsVault*.

Wrong comment for rewardsEndTimestamp

It is not **rewardsEndTimestamp** but **rewardsPerSecond** for which a zero value means "instant" reward payouts.

```
struct RewardInfo {
     uint8 decimals; // Reward Token Decimals
-    uint48 rewardsEndTimestamp; // {s} Rewards End Timestamp; 0 = instant
+    uint48 rewardsEndTimestamp; // {s} Rewards End Timestamp
     uint48 lastUpdatedTimestamp; // {s} Last updated timestamp
-    uint256 rewardsPerSecond; // {qRewardTok/s} Rewards per Second
+    uint256 rewardsPerSecond; // {qRewardTok/s} Rewards per Second; 0 = instant
     uint256 index; // {qRewardTok} Last updated reward index
     uint256 ONE; // {qRewardTok} Reward Token Scalar
 }
```

changeRewardSpeed() function should emit RewardInfoUpdate event

The **RewardInfoUpdate** event, according to its name and definition, should be emitted when **rewardsPerSecond** or **rewardsEndTimestamp** in the **RewardInfo** struct changes.

The event is missing from the *GenericMultiRewardsVault.changeRewardSpeed()* function.

```
         rewardInfos[rewardToken].rewardsPerSecond = rewardsPerSecond;
         rewardInfos[rewardToken].rewardsEndTimestamp = rewardsEndTimestamp;
+
+        emit Events.RewardInfoUpdate(rewardToken, rewardsPerSecond,
rewardsEndTimestamp);
     }
```

SCALING_FACTOR can be removed from GenericStakedAppreciatingVault

The calculations involving **SCALING_FACTOR** can be simplified, and the variable is no longer needed.

```
@@ -4,8 +4,6 @@ pragma solidity 0.8.24;
 import { ERC4626, IERC20, ERC20 } from
"@openzeppelin/contracts/token/ERC20/extensions/ERC4626.sol";
 import { SafeERC20 } from "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol";

-uint256 constant SCALING_FACTOR = 1e18;
-
 struct RewardTracker {
     uint256 rewardPeriodStart; // {s}
     uint256 rewardPeriodEnd; // {s}
@@ -113,12 +111,8 @@ contract GenericStakedAppreciatingVault is ERC4626 {
         uint256 previousDistributionPeriod = rewardTracker.rewardPeriodEnd -
rewardTracker.rewardPeriodStart;
         uint256 timePassed = block.timestamp - rewardTracker.rewardPeriodStart;

-        // D18{1} = {s} * D18 / {s}
-        uint256 timePassedPercentage = (timePassed * SCALING_FACTOR) /
previousDistributionPeriod;
-
-        // {qAsset} = {qAsset} * D18{1} / D18
-        uint256 accountedRewards = (rewardTracker.rewardAmount *
timePassedPercentage) / SCALING_FACTOR;
-        return accountedRewards;
+        // {qAsset} = {qAsset} * {s} / {s}
+        return rewardTracker.rewardAmount * timePassed / previousDistributionPeriod;
     }
```

Rewards only need to be accrued for one address in GenericMultiRewardsVautl

ERC4626 token balances in *GenericMultiRewawrdsVault* are non-transferrable and can only be changed upon deposits and withdrawals. For deposits and withdrawals, only the balance of one address changes. As a result, it is only necessary to accrue rewards for one address at a time. The recommendation is to keep the code as simple as possible and to remove the overhead of accruing rewards for two addresses.

```
@@ -51,7 +51,7 @@ contract GenericMultiRewardsVault is ERC4626, Ownable {
         address receiver,
         uint256 assets,
         uint256 shares
-    ) internal override accrueRewards(caller, receiver) {
+    ) internal override accrueRewards(receiver) {
         super._deposit(caller, receiver, assets, shares);
     }

@@ -61,7 +61,7 @@ contract GenericMultiRewardsVault is ERC4626, Ownable {
         address owner_,
         uint256 assets,
         uint256 shares
-    ) internal override accrueRewards(owner_, receiver) {
+    ) internal override accrueRewards(owner_) {
         super._withdraw(caller, receiver, owner_, assets, shares);
     }

@@ -86,7 +86,7 @@ contract GenericMultiRewardsVault is ERC4626, Ownable {
      * @param _rewardTokens Array of rewardTokens for which rewards should be
claimed.
```

```
      * @dev This function will revert if any of the rewardTokens have zero rewards
accrued.
      */
-    function claimRewards(address user, IERC20[] memory _rewardTokens) external
accrueRewards(msg.sender, user) {
+    function claimRewards(address user, IERC20[] memory _rewardTokens) external
accrueRewards(user) {
         for (uint8 i; i < _rewardTokens.length; i++) {
             uint256 rewardAmount = accruedRewards[user][_rewardTokens[i]];

@@ -278,7 +278,7 @@ contract GenericMultiRewardsVault is ERC4626, Ownable {
         return rewardTokens;
     }

-    modifier accrueRewards(address _caller, address _receiver) {
+    modifier accrueRewards(address _user) {
         IERC20[] memory _rewardTokens = rewardTokens;
         for (uint256 i; i < _rewardTokens.length; i++) {
             IERC20 rewardToken = _rewardTokens[i];
@@ -288,13 +288,7 @@ contract GenericMultiRewardsVault is ERC4626, Ownable {
                 _accrueRewards(rewardToken, _accrueStatic(rewards));
             }

-            _accrueUser(_receiver, rewardToken);
-
-            // If a deposit/withdraw operation gets called for another user we should
-            // accrue for both of them to avoid potential issues
-            if (_receiver != _caller) {
-                _accrueUser(_caller, rewardToken);
-            }
+            _accrueUser(_user, rewardToken);
         }
         _;
     }
```

Centralization risks

Owner in GenericMultiRewardsVault is fully trusted

In the initial audit commit, the **owner** role in *GenericMultiRewardsVault* can lock all funds in the contract, including the underlying **assets**. This can be achieved by adding a malicious **rewardToken** with a very large supply such as to make the *accrueRewards()* modifier revert.

By introducing the *ragequit()* function, the **owner** cannot prevent the users from withdrawing their underlying **assets**. As a result, the **owner** is only fully trusted with regards to the **rewardTokens**. A malicious **owner** can cause a permanent loss of all the rewards in the contract, including rewards that have already been accrued but not claimed.

Distributor in GenericMultiRewardsVault can change reward speeds

The *GenericMultiRewardsVault* contract assigns one **distributor** per **rewardToken**. This role is allowed to change the reward speed of the **rewardToken**. It has no privileges beyond changing the reward speed and the contract **owner** can remove its privileges.

Still, being able to change the reward speed is a trusted action. A malicious **distributor** could be able to earn most of the reward itself by paying out all rewards in one second and making a large flash-deposit.

Systemic risks

External reward tokens risk

There is no guarantee that the reward tokens paid out in *GenericMultiRewardsVault* will retain their value or are even valuable in the first place. The tokens are added by the **owner** of the contract who should ensure a high quality of reward tokens.

Reward tokens that are paid out instantly are vulnerable to flash deposits

GenericMultiRewardsVault supports two modes for paying out rewards. In the first mode, a certain amount of the **rewardToken** is paid out each second. In the second mode, all the rewards that are sent to the contract are paid out instantly.

An attacker can possibly trigger a reward payout in an external contract that then sends rewards to *GenericMultiRewardsVault* and perform a sandwich attack. Since the attacker can perform the attack in a single transaction, they can leverage flash-loans.

This is a known risk, and third-party integrations need to ensure instant payouts cannot be exploited.