Security Review Report
NM-0443-Royco Vaults

NETHERMIND
SECURITY

(Feb 20, 2025)

Contents

1 Executive Summary

This document presents the security review performed by Nethermind Security for Boring Vault Chef smart contracts. The Royco team has forked the `BoringVault` contract and implemented changes to facilitate the distribution of rewards in multiple tokens and claims. The `BoringChef` contract, inherited by `Boring Vault` implements the rewards distribution logic, along with management of epochs and user balances.

This security review focuses exclusively on the smart contracts listed in Section 2 (*Audited Files*).

The audit was performed using (a) manual analysis of the codebase and (b) creation of test cases. **Along this document, we report** eight points of attention, where two are classified as `Critical`, one is classified as `Medium`, one is classified as `Low`, and four are classified as `Informational` and `Best Practices`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the compilation, tests, and automated tests. Section 9 concludes the document.

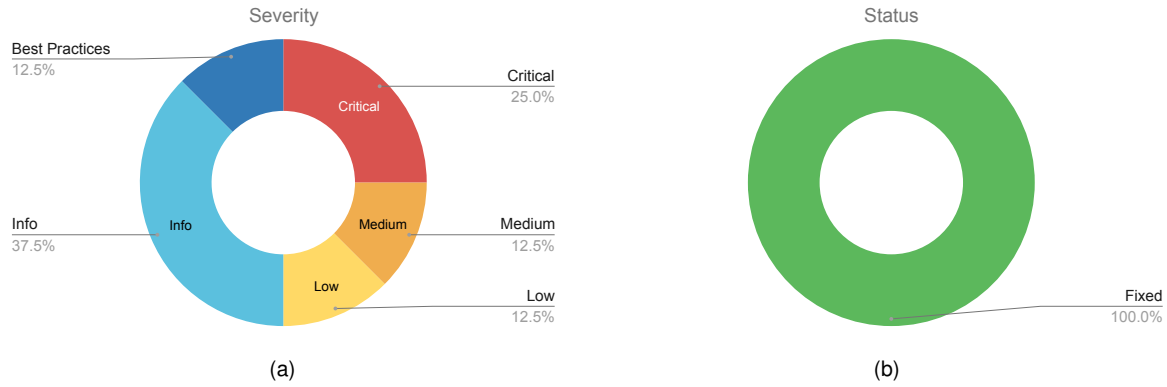

Fig. 1: Distribution of issues: Critical (2), **High** (0), **Medium** (1), **Low** (1), **Undetermined** (0), **Informational** (3), **Best Practices** (1). **Distribution of status: Fixed** (8), **Acknowledged** (0), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Feb 19, 2025
Response from Client	Regular responses during audit engagement
Final Report	Feb 20, 2025
Repository	boring-vault-chef
Commit (Audit)	9591cffef3c2c7f71e28b6fdfce9d905300b1be3
Commit (Final)	fe71b4267fb2414db3473fb8fe7333cd7d035cc2
Documentation Assessment	High
Test Suite Assessment	Medium

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	src/base/BoringVault.sol	72	46	63.9%	27	145
2	src/boring-chef/BoringChef.sol	380	235	61.8%	100	715
3	src/boring-chef/BoringSafe.sol	10	9	90.0%	3	22
	Total	**462**	**290**	**62.8%**	**130**	**882**

3 Summary of Issues

	Finding	Severity	Update
1	Calling `claimRewards` with an empty array allows reclaiming rewards	Critical	Fixed
2	Missing cache initialization leads to double claiming of rewards	Critical	Fixed
3	Rewards including epochs with no eligible shares cannot be claimed	Medium	Fixed
4	Incorrect `totalSharesBalance` update leads to an underflow error	Low	Fixed
5	Epoch eligible shares can be incorrectly increased	Info	Fixed
6	Incorrect epoch in the `UserWithdrawnFromEpoch` event	Info	Fixed
7	`claimRewards` allows claiming invalid reward Ids	Info	Fixed
8	Unused and redundant code	Best Practices	Fixed

4 System Overview

The Royco team developed a fork of the Boring Vault, which uses the Boring Chef to facilitate retroactive reward distribution and claims. The rewards can be in any token and distributed over any historical duration. Depositors who are eligible for these rewards, based on their participation during the specified duration, can claim them from the BoringChef contract.

The system is comprised of three primary smart contracts:

- **BoringChef**: The core contract responsible for tracking depositors' historical shares across epochs and handling reward distribution logic.
- **BoringVault**: A contract that inherits from the `BoringChef` contract, exposing the main external vault functions, including manager operations and the `enter` and `exit` functions, it outsources complex functionality to external contracts.
- **BoringSafe**: This contract holds the funds committed for reward distribution within the `BoringChef` contract.

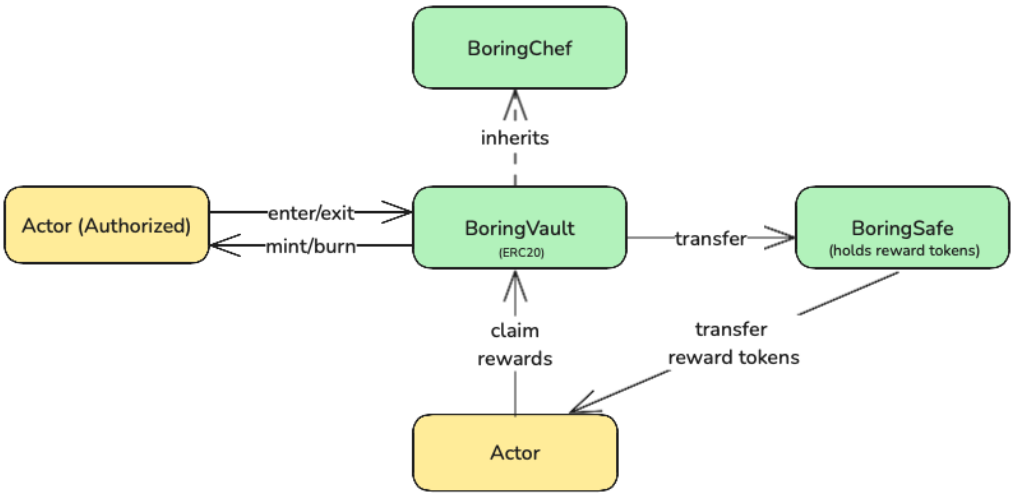

Fig. 2: Royco Vaults overview

4.1 Epoch Management

An epoch represents a defined time period in which user balances are recorded. Each epoch has a start timestamp, an end timestamp, and a total amount of eligible shares, which excludes any funds that do not accrue rewards. The total eligible shares for an epoch differ from the total supply, as the latter includes non-eligible shares.

The manager can transition to the next epoch using the `rollOverEpoch` function:

```
function rollOverEpoch() external requiresAuth
```

This function is invoked through the `manage` function defined in the `BoringVault` contract. It transfers eligible shares from the current epoch to the next one.

4.2 Enter and Exit Vault Flows

The `enter` function allows authorized addresses to deposit ERC20 tokens into the vault. In exchange, the function mints a specified amount of vault shares and assigns them to the receiver address (`to`).

```
function enter(address from, ERC20 asset, uint256 assetAmount, address to, uint256 shareAmount)
        external
        requiresAuth
```

When tokens are deposited, the user's participation in the next epoch is increased based on the minted shares. The next epoch's eligible shares are also incremented by the same amount. If the user has disabled reward accrual, stored share amount will not be updated.

The `exit` function allows authorized addresses to burn vault shares of a specified user and withdraw a corresponding amount of ERC20 tokens.

```
function exit(address to, ERC20 asset, uint256 assetAmount, address from, uint256 shareAmount)
        external
        requiresAuth
```

When shares are burned, the user's participation in the next epoch is reduced. This logic is implemented through the `_decreaseCurrentAndNextEpochParticipation` function, which handles multiple cases: it first attempts to withdraw from the next epoch and, if insufficient, withdraws from the current epoch. The total eligible shares for each epoch are adjusted accordingly. If the user reward accrual is disabled, no updates occur.

4.3 Rewards Distribution and Claims

Rewards are distributed by authorized actors within the vault. Each reward is defined by a start and end epoch, a reward token, and a reward rate (the amount of tokens distributed per second). The `distributeRewards` function enables the distribution of a set of rewards:

```
function distributeRewards(
    address[] calldata tokens,
    uint256[] calldata amounts,
    uint48[] calldata startEpochs,
    uint48[] calldata endEpochs
) external requiresAuth
```

Rewards are assigned unique incremental IDs and can only be distributed for past epochs. The reward amounts are transferred to the `BoringSafe` contract.

Users can claim their eligible rewards using the `claimRewards` function by providing an array of reward IDs to claim.

```
function claimRewards(uint256[] calldata rewardIds) external
```

For each reward ID, the user's owed amount is calculated based on the ratio of shares they held during a specific epoch compared to the total eligible shares. The owed amounts for the epochs covered by the reward are summed up and transferred from the `BoringSafe` to the user.

Rewards are grouped into buckets, each containing up to 256 rewards. The `userToRewardBucketToClaimedRewards` mapping stores a 256-bit field for each bucket, where the bit at the i-th position indicates whether the user has claimed the reward ID at that position in the bucket.

```
mapping(address user => mapping(uint256 rewardBucket => uint256 claimedRewards)) public
        userToRewardBucketToClaimedRewards;
```

The vault allows authorized addresses to enable or disable reward accrual for a user. Enabling rewards increases both the user's balance update and the eligible shares for the next epoch by their current shares balance.

```
function enableRewardAccrual(address user) external requiresAuth
```

Disabling reward accrual reduces the user's balance update and total eligible shares for the next (or current) epoch by their current balance.

```
function disableRewardAccrual(address user) external requiresAuth
```

5 Risk Rating Methodology

The risk rating methodology used by Nethermind Security follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind Security also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [Critical] Calling `claimRewards` with an empty array allows reclaiming rewards

File(s): BoringChef.sol

Description: The `claimRewards` function accepts an array of `rewardIds` to claim. It calls the internal function `_getEpochRangeForRewards` to determine the epoch range for each reward and fetch the corresponding Reward structs. Additionally, this function updates the `userToRewardBucketToClaimedRewards` mapping to mark rewards as claimed by modifying the appropriate bit in the reward bucket.

```
function _getEpochRangeForRewards(uint256[] calldata rewardIds)
    internal
    returns (uint48 minEpoch, uint48 maxEpoch, Reward[] memory rewardsToClaim)
{
    // ...
    // Variables to cache reward claim data as a gas optimization.
    uint256 cachedRewardBucket;
    uint256 cachedClaimedRewards;

    for (uint256 i = 0; i < rewardsLength; ++i) {
        // ...
    }

    // @audit If rewardIds.length is 0, then it would unset all bits for 0th reward bucket.
    // Write back the final cache to persistent storage
    userToRewardBucketToClaimedRewards[msg.sender][cachedRewardBucket] = cachedClaimedRewards;
}
```

However, when the function is called with an empty `rewardIds` array, the loop is skipped, and the final line is executed. This results in all bits in the `userToRewardBucketToClaimedRewards` mapping for the 0th reward bucket being unset, effectively marking them as unclaimed. Consequently, users can reclaim rewards in the 0th reward bucket multiple times.

Recommendation(s): Consider updating the `_getEpochRangeForRewards` function to revert when the `rewardIds` array is empty.

Status: Fixed

Update from the client: 700addd3971d36b1d807bd3e43d7b9132c76b577

6.2 [Critical] Missing cache initialization leads to double claiming of rewards

File(s): BoringChef.sol

Description: The `claimRewards` function takes a `rewardIds` array as an argument and calls `_getEpochRangeForRewards` to get the epoch range for all rewards to claim and the corresponding Reward structs. A nested mapping `userToRewardBucketToClaimedRewards` is used to efficiently keep track of claimed rewards per user. Each reward bucket contains batches of 256 contiguous `rewardIds`. Each bit corresponds to whether the reward is claimed or not. To optimize gas usage, the `_getEpochRangeForRewards` function caches reward bucket data, minimizing the number of `SLOAD` and `SSTORE` operations when claiming rewards

```
function _getEpochRangeForRewards(uint256[] calldata rewardIds)
    internal
    returns (uint48 minEpoch, uint48 maxEpoch, Reward[] memory rewardsToClaim)
{
    // ...

    // Variables to cache reward claim data as a gas optimization.
    uint256 cachedRewardBucket;
    uint256 cachedClaimedRewards;

    for (uint256 i = 0; i < rewardsLength; ++i) {
        // Cache management (reading and writing)
        {
            uint256 rewardBucket = rewardIds[i] / 256;

            if (i == 0) {
                // @audit cachedRewardBucket is not initialized
                cachedClaimedRewards = userToRewardBucketToClaimedRewards[msg.sender][rewardBucket];
            } else if (cachedRewardBucket != rewardBucket) {
                // @audit invalid claim data written into storage
                userToRewardBucketToClaimedRewards[msg.sender][cachedRewardBucket] = cachedClaimedRewards;

                // Updated cache with the new reward bucket and rewards bit field
                cachedRewardBucket = rewardBucket;
                cachedClaimedRewards = userToRewardBucketToClaimedRewards[msg.sender][rewardBucket];
            }
            // ...
    }
}
```

However, the caching mechanism contains an issue. Specifically, during the first iteration of the loop (i = 0), only `cachedClaimedRewards` is set, but `cachedRewardBucket` is not. As a result, `cachedClaimedRewards` may contain claim statuses from any reward bucket, while `cachedRewardBucket` defaults to 0.

This causes an inconsistency: if the `rewardId` in the first iteration belongs to a reward bucket other than 0 (i.e., rewardId > 255), the subsequent iteration will incorrectly store the updated claim data in bucket 0, enabling users to claim the same reward multiple times for any rewardId greater than 255, by only passing one `rewardId` in the argument.

Recommendation(s): Consider assigning `cachedRewardBucket` in the first iteration of the loop.

Status: Fixed.

Update from the client: 1faf594d0dabd64fefd9440153b0c37f7ed16f05

6.3 [Medium] Rewards including epochs with no eligible shares cannot be claimed

File(s): BoringChef.sol

Description: The `claimRewards` function takes an array of `rewardIds` and calls `_getEpochRangeForRewards` to fetch the epoch range for all rewards to claim, along with the corresponding Reward structs. This epoch range is then passed to the `_computeUserShareRatiosAndDurations` function, which calculates the user's share ratios and epoch durations for each epoch within the specified range from `minEpoch` to `maxEpoch`.

```
function _computeUserShareRatiosAndDurations(
    uint48 minEpoch,
    uint48 maxEpoch,
    BalanceUpdate[] storage userBalanceUpdates
) internal view returns (uint256[] memory userShareRatios, uint256[] memory epochDurations) {
    uint256 epochCount = maxEpoch - minEpoch + 1;
    userShareRatios = new uint256[](epochCount);
    epochDurations = new uint256[](epochCount);

    // ...

    // Loop over each epoch from minEpoch to maxEpoch.
    for (uint256 epoch = minEpoch; epoch <= maxEpoch; ++epoch) {
        // ...

        // Retrieve the epoch data.
        Epoch storage epochData = epochs[uint48(epoch)];
        uint256 epochIndex = epoch - minEpoch;
        // Calculate the user's fraction of shares for this epoch.
        // @audit It will revert for epochData.eligibleShares equals to 0
        userShareRatios[epochIndex] = epochSharesBalance.divWadDown(epochData.eligibleShares);
        // Calculate the epoch duration.
        epochDurations[epochIndex] = epochData.endTimestamp - epochData.startTimestamp;
    }
}
```

However, an issue arises when calculating the user's share ratios. The function divides the user's shares by the total eligible shares for each epoch, causing the transaction to revert if `eligibleShares` is zero. Since rewards are distributed across multiple epochs, if any epoch in the range has zero eligible shares, users cannot claim rewards for that `rewardId`.

Recommendation(s): Implement a check to handle the case where `eligibleShares` for an epoch is `0` in `_computeUserShareRatiosAndDurations`.

Status: Fixed.

Update from the client: Fix reversions: a4f768c398c269298c1a51b0e9e8875ddebfd2f6

6.4 [Low] Incorrect `totalSharesBalance` update leads to an underflow error

File(s): `BoringChef.sol`

Description: The `_decreaseCurrentAndNextEpochParticipation` function implements the logic to decrease the user's participation for the current or next epoch. This function is invoked in several cases: when a user disables reward accrual, when a share transfer occurs, or when a user exits the vault and burns a specified number of shares. It is designed to adjust the user's participation for the current or next epoch accordingly.

However, instead of decreasing the balance by the specified amount, the function overrides the stored shares balance with the user's current ERC20 balance. This logic can lead to inaccuracies, particularly when the function is triggered by the `disableRewardAccrual` flow. In such cases, the contract should set the user balance update to zero (i.e., reducing the balance update by the full balance), while the user's ERC20 balance remains unchanged. The current code mistakenly overrides the balance with the user's ERC20 balance, leading to potential underflow errors.

```
function _decreaseCurrentAndNextEpochParticipation(address user, uint128 amount) internal {// ...int128 userBalance =
↪  uint128(balanceOf[user]);
// ...
// CASE 1: No deposit for the next epoch.
if (lastBalanceUpdate.epoch <= currEpoch) {
    if (lastBalanceUpdate.epoch == currEpoch) {
        // @audit totalSharesBalance set to current user ERC20 balance
        lastBalanceUpdate.totalSharesBalance = userBalance;
    } else {
        // @audit totalSharesBalance set to current user ERC20 balance
        userBalanceUpdates.push(BalanceUpdate({epoch: currEpoch, totalSharesBalance: userBalance}));
    }
    // ...
}

// CASE 2: Only deposit made is for the next epoch.
if (balanceUpdatesLength == 1) {
    // @audit totalSharesBalance set to current user ERC20 balance
    lastBalanceUpdate.totalSharesBalance = userBalance;
    // ...
}

// @audit underflow error occures due to incorrect balanceUpdate value
uint128 amountDepositedIntoNextEpoch =
    lastBalanceUpdate.totalSharesBalance - secondLastBalanceUpdate.totalSharesBalance;

// CASE 3: Deposit Made for the next epoch and the full withdrawal amount cannot be removed solely from the last update.
if (amount > amountDepositedIntoNextEpoch) {
      uint128 amountToWithdrawFromCurrentEpoch = (amount - amountDepositedIntoNextEpoch);
    if (secondLastBalanceUpdate.epoch == currEpoch) {... } else {
        // ...
        // @audit  totalSharesBalance set to current user ERC20 balance
        // ...
    }
    // ...
} else {...}
}
```

Recommendation(s): Consider updating the `_decreaseCurrentAndNextEpochParticipation` function logic to decrease the stored balance update by the provided amount instead of relying on the user's current ERC20 balance.

Status: Fixed

Update from the client: 2ba2c329ffff330063a348d67fe98a4f0b8c30f5

Update from Nethermind Security: The following line incorrectly reduces the `lastBalanceUpdate.totalSharesBalance` by `amountToWithdrawFromCurrentEpoch`, it should be reduced by the full `amount` instead. This is because the `lastBalanceUpdate` contains the shares amount of the next epoch.

Update from the client: fe71b4267fb2414db3473fb8fe7333cd7d035cc2

6.5 [Info] Epoch eligible shares can be incorrectly increased

File(s): BoringChef.sol

Description: The `enableRewardAccrual` function allows rewards to be enabled for a specified user. It increases the user's next epoch participation based on their current balance, effectively enabling reward accrual starting from the next epoch. However, the function lacks a check to determine whether the user's rewards accrual is already enabled, allowing the function to be called multiple times.

```
function enableRewardAccrual(address user) external requiresAuth {
    //@audit Absence of check if rewards are enabled already
    addressToIsDisabled[user] = false;
    _increaseNextEpochParticipation(user, uint128(balanceOf[user]));
}
```

The internal `_increaseNextEpochParticipation` function updates the user's `totalSharesBalance` using their current ERC20 balance and increases the next epoch's eligible shares by the provided amount. The vulnerability allows repeatedly calling the function and artificially increasing the next epoch's eligible shares, reducing other users' share ratios and preventing them from claiming their eligible rewards.

```
function _increaseNextEpochParticipation(address user, uint128 amount) internal {
    // ...
    // Get the epoch data for the current epoch and next epoch (epoch to deposit for)
    Epoch storage epochData = epochs[nextEpoch];
    // @audit the epoch `eligibleShares` can be incorrectly increased
    epochData.eligibleShares += amount;

    // Emit event for this deposit
    emit UserDepositedIntoEpoch(user, nextEpoch, amount);
}
```

Recommendation(s): Implement a check in the `enableRewardAccrual` function to prevent it from being called if rewards accrual is already enabled for the user. It is also recommended to implement a similar check in the `disableRewardAccrual` function.

Status: Fixed

Update from the client: 05c7db35cade4cd6ab909769a1d9304ba0c785d2

6.6 [Info] Incorrect epoch in the `UserWithdrawnFromEpoch` event

File(s): BoringChef.sol

Description: The `_decreaseCurrentAndNextEpochParticipation` function emits the `UserWithdrawnFromEpoch` event, indicating the withdrawn amount from each epoch for the specified user. However, in the 4th case of the function, where the full withdrawal amount is taken from the next epoch, the event erroneously refers to the current epoch (`currEpoch`) instead of the next epoch (`nextEpoch`).

```
    function _decreaseCurrentAndNextEpochParticipation(address user, uint128 amount) internal {
        // ...
         // CASE 4: The full amount can be withdrawn from the next epoch. Modify the next epoch (last) update.
            lastBalanceUpdate.totalSharesBalance -= amount;
            epochs[uint48(nextEpoch)].eligibleShares -= amount;
            // Emit event for this withdrawal.
            emit UserWithdrawnFromEpoch(user, currEpoch, amount); //@audit must be `nextEpoch` instead of `currEpoch`
            return;
        }
    }
```

Recommendation(s): Update the event argument to use the correct epoch: `nextEpoch` instead of `currEpoch`.

Status: Fixed

Update from the client: 4d2ceca484f25405a84942f835a4a8283bad9cca

6.7 [Info] `claimRewards` allows claiming invalid reward Ids

File(s): `BoringChef.sol`

Description: The `claimRewards` function enables users to claim rewards by providing an array of reward IDs. Each claimed reward's bit is marked as 1 in the `claimedRewards` bit field variable, preventing double claims. However, the function does not validate the provided reward IDs. If a user inputs a reward ID that exceeds the `maxRewardId`, the corresponding bit is set to true, effectively blocking future claims for that reward.

Recommendation(s): Implement a validation check within the `claimRewards` function to ensure that each provided reward ID is valid.

Status: Fixed

Update from the client: d4df93f88372a006a77f17cbaa70644b7db95ac6

6.8 [Best Practices] Unused and redundant code

File(s): `BoringChef.sol`

Description: We provide a non-exhaustive list of unused code below:

- The `RewardClaimInfo` struct is defined in the `BoringChef` contract but has never been used in the code;

```
/// @dev A record of a user's balance changing at a specific epoch
struct RewardClaimInfo {
    /// @dev The epoch in which the deposit was made
    uint48 epoch;
    /// @dev The total number of shares the user has at this epoch
    uint128 totalSharesBalance;
}
```

- The `claimRewards` function processes multiple reward IDs by grouping rewards into unique tokens and amounts, minimizing the number of `transfer` calls to save gas. It initializes arrays for `uniqueTokens` and `tokenAmounts`, using `uniqueCount` to track their lengths. A check ensures that reward campaigns with zero rewards are skipped. The function then loops over the tokens to transfer the accumulated rewards. However, the check `if (tokenAmounts[i] > 0)` before transferring is redundant, as rewards with zero owed have already been excluded;

```
function claimRewards(uint256[] calldata rewardIds) external {
    // ...
    for (uint256 i = 0; i < rewardIds.length; ++i) {
        // Calculate the total rewards owed for this reward campaign.
        uint256 rewardsOwed = _calculateRewardsOwed(rewardsToClaim[i], minEpoch, userShareRatios, epochDurations);

        if (rewardsOwed > 0) {
            // ...
            if (!found) {
                uniqueTokens[uniqueCount] = rewardsToClaim[i].token;
                tokenAmounts[uniqueCount] = rewardsOwed;
                uniqueCount++;
            }
            // Emit the reward-claim event per reward campaign.
            emit UserRewardsClaimed(msg.sender, rewardIds[i], rewardsOwed);
        }
    }

    // Finally, do one transfer per unique reward token.
    for (uint256 i = 0; i < uniqueCount; ++i) {
        // @audit Redundant check
        if (tokenAmounts[i] > 0) {
            boringSafe.transfer(uniqueTokens[i], msg.sender, tokenAmounts[i]);
        }
    }
}
```

Recommendation(s): Remove the unused and redundant code.

Status: Fixed

Update from the client: Fixed in commit 5e03120068cc374ed1b0f460afc35cf548aaf700 and accc265991239dbfd5c665d105ae4ed7972af95e

7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about Royco Vaults documentation

The documentation for the Royco Vaults smart contracts is contained in the README file in the project's GitHub repository. This file provides an overview of the Boring Vault Chef architecture and outlines the implemented changes to facilitate reward distribution and claims.

The functions are thoroughly documented. Each function includes detailed comments that explain the logic behind its implementation and the significance of each parameter.

Additionally, the `Royco` team has provided a comprehensive walkthrough of the project and has addressed all questions and concerns raised by the Nethermind Security team during their regular calls.

8 Test Suite Evaluation

The test suite for `Royco Vaults` smart contracts covers the `BoringChef` public functionalities. It includes main flows such as entering and exiting the vault, reward distribution and claims, and share transfers. Foundry's fuzzing feature is utilized throughout the test suite to verify that functions handle a wide variety of inputs correctly. However, the test suite could be further enhanced by testing cases with large reward IDs and multiple buckets. In addition, more fuzzing tests could be added to explore different scenarios.

```
> forge test --match-contract BoringChefTest
[] Compiling...
[] Compiling 200 files with Solc 0.8.21
[] Solc 0.8.21 finished in 113.93s
Compiler run successful!

Warning: `testFail*` has been deprecated and will be removed in the next release. Consider changing to
 ↪  test_Revert[If|When]_Condition and expecting a revert. Found deprecated testFail* function(s):
 ↪  testFailClaimRewardsAlreadyClaimed, testFailDistributeRewardsEndEpochInFuture,
 ↪  testFailRewardsStartEpochGreaterThanEndEpoch, testFailTransferExceedingBalance, testFail_WithdrawExceedingBalance.
Ran 34 tests for test/BoringChef.t.sol:BoringChefTest
[PASS] testBasicTransfer(uint256,uint256) (runs: 256, : 512297, ~: 512297)
[PASS] testBlacklistUser() (gas: 1373)
[PASS] testComplexRewardClaiming() (gas: 21260030)
[PASS] testComplexRewardDistribution() (gas: 3385237)
[PASS] testDistributeRewardsValidRange() (gas: 2520015)
[PASS] testFailClaimRewardsAlreadyClaimed() (gas: 664938)
[PASS] testFailDistributeRewardsEndEpochInFuture() (gas: 102298)
[PASS] testFailRewardsStartEpochGreaterThanEndEpoch() (gas: 159263)
[PASS] testFailTransferExceedingBalance() (gas: 320829)
[PASS] testFail_WithdrawExceedingBalance() (gas: 329640)
[PASS] testFindUserBalanceAtEpochAllUpdatesAfter() (gas: 330489)
[PASS] testFindUserBalanceAtEpochExactMatch() (gas: 363008)
[PASS] testFindUserBalanceAtEpochMultipleUpdates() (gas: 535147)
[PASS] testFindUserBalanceAtEpochNoDeposits() (gas: 93885)
[PASS] testFuzz_DepositWithdraw_Random(uint256,uint256,uint256) (runs: 256, : 1672825, ~: 1565457)
[PASS] testFuzz_DepositWithdraw_SameEpoch_ExistingBalance(uint256,uint256) (runs: 256, : 709234, ~: 709234)
[PASS] testFuzz_DepositWithdraw_SameEpoch_UpcomingUpdate(uint256) (runs: 256, : 307670, ~: 307670)
[PASS] testFuzz_Deposit_RollOverMultipleEpochs_Withdraw(uint256,uint256,uint256) (runs: 256, : 416030, ~: 418391)
[PASS] testFuzz_Deposit_RollOver_Deposit_Withdraw(uint256,uint256) (runs: 256, : 450508, ~: 450508)
[PASS] testFuzz_Deposit_RollOver_Withdraw(uint256,uint256) (runs: 256, : 337959, ~: 337959)
[PASS] testFuzz_MultipleDeposits(uint256,uint256) (runs: 256, : 348322, ~: 348322)
[PASS] testFuzz_MultipleDeposits_DifferentEpochs(uint256,uint256) (runs: 256, : 400426, ~: 400426)
[PASS] testFuzz_Withdraw(uint256) (runs: 256, : 292831, ~: 292830)
[PASS] testFuzz_Withdraw_MultipleEpochsAndDeposits(uint256) (runs: 256, : 535377, ~: 535377)
[PASS] testManualEpochRollover() (gas: 312449)
[PASS] testMultipleEpochRollovers() (gas: 458237)
[PASS] testPrintBalanceUpdates((uint48,uint128)[]) (runs: 256, : 501610, ~: 489731)
[PASS] testRewardClaiming() (gas: 3680247)
[PASS] testSingleEpochRewardDistribution() (gas: 1136581)
[PASS] testUser() (gas: 3235)
[PASS] test_MultipleDeposits() (gas: 269027)
[PASS] test_SingleDeposit() (gas: 245411)
[PASS] test_WithdrawAll() (gas: 278853)
[PASS] test_WithdrawPartial() (gas: 331824)
Suite result: ok. 34 passed; 0 failed; 0 skipped; finished in 3.18s (17.40s CPU time)

Ran 1 test suite in 3.19s (3.18s CPU time): 34 tests passed, 0 failed, 0 skipped (34 total tests)
```

8.0.1 Slither

All the relevant issues raised by Slither have been incorporated into the issues described in this report.

8.0.2 AuditAgent

All the relevant issues raised by the AuditAgent have been incorporated into this report. The AuditAgent is an AI-powered smart contract auditing tool that analyses code, detects vulnerabilities, and provides actionable fixes. It accelerates the security analysis process, complementing human expertise with advanced AI models to deliver efficient and comprehensive smart contract audits. Available at https://app.auditagent.nethermind.io.

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.