Nest Credit

Security Assessment

January 15th, 2025 — Prepared by OtterSec

Nicholas R. Putra nicholas@osec.io

Renato Eugenio Maria Marziano renato@osec.io

Robert Chen r@osec.io

Table of Contents

01 — Executive Summary

Overview

Plume Network engaged OtterSec to assess the `nest-staking-contracts` program. This assessment was conducted between December 18th, 2024 and January 6th, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 7 findings throughout this audit engagement.

In particular, we identified a high-risk vulnerability in the logic for buying vault tokens, which risks misallocated approvals by granting access to the teller instead of the vault and relies on untrusted user-provided addresses (OS-PNT-ADV-01). Additionally, `AggregateToken` lacks the functionality to trigger the `notifyRedeem` method, resulting in unprocessed asynchronous redemption requests despite the enabling of the `asyncRedeem` feature by default (OS-PNT-ADV-00). Moreover, there are rounding-down issues during share calculations in the withdraw functionality, potentially allowing users to withdraw more assets than justified by the shares burned, resulting in fund imbalances and losses in the vault (OS-PNT-SUG-01).

We also made suggestions to ensure adherence to coding best practices (OS-PNT-SUG-00).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/plumenetwork/contracts. This audit was performed against commit d0fb773.

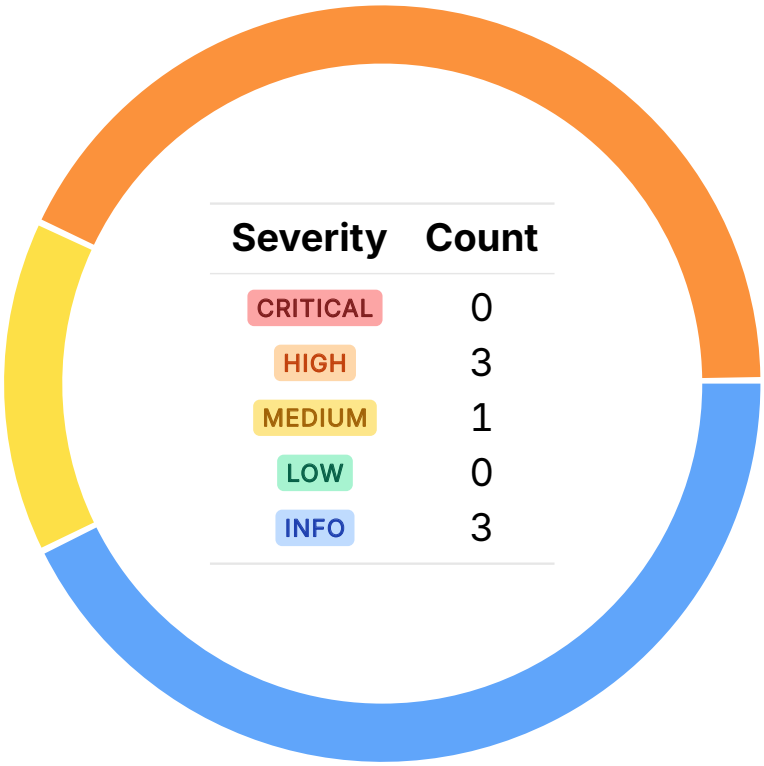
A brief description of the program is as follows:

Name	Description
nest-staking-contracts	These contracts support the Nest Staking product, allowing users to permissionlessly deposit and withdraw from the Nest Staking vault. Users can exchange stablecoins such as $USDC and $USDT for shares in the vault, represented by $NEV.

03 — Findings

Overall, we reported 7 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	0
HIGH	3
MEDIUM	1
LOW	0
INFO	3

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-PNT-ADV-00	HIGH	RESOLVED	`AggregateToken` lacks the functionality to trigger `_notifyRedeem`, resulting in unprocessed asynchronous redemption requests despite the `asyncRedeem` feature being enabled by default.
OS-PNT-ADV-01	HIGH	RESOLVED	`buyVaultToken` risks misallocated approvals by granting access to the teller instead of the vault and by relying on untrusted user-provided addresses, potentially allowing share inflation and asset recovery exploitation.
OS-PNT-ADV-02	HIGH	RESOLVED	`ComponentToken` incorrectly uses wrong conversion rate for synchronous `deposit` and `redeem` operations due to ratio of supply and underlying assets changing prematurely.
OS-PNT-ADV-03	MEDIUM	RESOLVED	`unfeatureToken` in `NestStaking` has a gas cost that increases linearly due to iterating over the `featuredList` to find and remove a token, resulting in high gas costs as the list grows substantially.

Unprocessed Async Redemption Requests HIGH OS-PNT-ADV-00

Description

The `asyncRedeem` mechanism allows users to request a redemption that is not processed instantly. Instead, the request is handled asynchronously, implying an external mechanism must later notify the system of its completion via `ComponentToken::_notifyRedeem`. It reduces the pending redemption request balance and marks the redeemed shares as claimable by the user.

However, `AggregateToken` never invokes `_notifyRedeem` despite having `asyncRedeem` enabled. This causes redemption requests to remain in the `pendingRedeemRequest` mapping indefinitely, and the `claimableRedeemRequest` mapping is never updated. As a result, users are unable to claim their assets or redeemed tokens.

Remediation

Ensure that `AggregateToken` calls `_notifyRedeem` to close the asynchronous redemption lifecycle.

Patch

Resolved in dbacef9.

Lack of Proper Access Control Logic in BuyVaultToken HIGH OS-PNT-ADV-01

Description

Multiple vulnerabilities arise from the `buyVaultToken` function within `AggregateToken` due to inconsistent access control logic. Currently, the function approves the `teller` contract to spend the tokens deposited by the user. However, the approval should be granted to the vault itself to ensure proper flow of assets. Moreover, the current implementation takes `_teller` as a parameter directly from the user, trusting it to be a legitimate `teller` contract. This allows a user to provide a dummy address that they control.

>_ nest/src/AggregateToken.sol SOLIDITY

```
function buyVaultToken(
    address token,
    uint256 assets,
    uint256 minimumMint,
    address _teller
) public nonReentrant returns (uint256 shares) {
    [...]
    ITeller teller = ITeller(_teller);
    // Verify deposit is allowed through teller
    if (teller.isPaused()) {
        revert TellerPaused();
    }
    if (!teller.isSupported(IERC20(token))) {
        revert AssetNotSupported();
    }
    // Transfer assets from sender to this contract
    SafeERC20.safeTransferFrom(IERC20(token), msg.sender, address(this), assets);
    // Approve teller to spend assets
    SafeERC20.forceApprove(IERC20(token), address(teller), assets);
    [...]
}
```

Allowing users to pass arbitrary addresses for crucial contracts like the `teller` introduces a vector for potential exploitation. If the off-chain calculation depends on the `totalAsset` held by the `AggregateToken`, users may pass a dummy `teller` address to inflate their share values. After redemption, if there is some lingering approval associated with the fraudulent `teller`, users may recover more assets than they should be entitled to. Additionally, `buyVaultToken` allows any user to invoke it, implying that it is not restricted by roles.

Remediation

Ensure that approval is granted to the `vault` instead of the `teller` within `buyVaultToken`, and refrain from trusting user-provided address inputs. Additionally, limit access to `buyVaultToken` so that only the `MANAGER_ROLE` may access it, as enforced in `buyComponentToken`.

Patch

Resolved in 223af8d.

Incorrect Asset-Share Conversion Order HIGH OS-PNT-ADV-02

Description

The `ComponentToken` has a feature that allows users to perform synchronous `redeem` and `deposit` actions.

However, when calculating the needed `shares` to burn, the current code incorrectly performs the `burn` before doing the conversion. This results in receiving more `assets` than intended, as the ratio is higher due to the premature `burn`.

>_ *nest/src/ComponentToken.sol* SOLIDITY

```
function redeem(
    uint256 shares,
    address receiver,
    address controller
) public virtual override(ERC4626Upgradeable, IERC7540) nonReentrant returns (uint256 assets) {
    [...]
    } else {
        // For sync redemptions, process normally
        _burn(controller, shares);
        assets = convertToAssets(shares);
    }
    [...]
}
```

A similar issue occurs in the `deposit` process, where the current code incorrectly transfers the user's assets before performing the conversion, resulting in fewer minted `shares` than intended.

>_ *nest/src/ComponentToken.sol* SOLIDITY

```
function deposit(
    uint256 assets,
    address receiver,
    address controller
) public virtual nonReentrant returns (uint256 shares) {
    [...]
    } else {
        SafeERC20.safeTransferFrom(IERC20(asset()), controller, address(this), assets);
        shares = convertToShares(assets);
    }
    [...]
}
```

Remediation

Ensure that the conversion occurs before any changes to the supply or the underlying assets.

Patch

Resolved in f0960c9.

Excessive Gas Cost for Execution of UnfeatureToken MEDIUM OS-PNT-ADV-03

Description

The issue demonstrates how users can manipulate `NestStaking::unfeatureToken` to consume an excessive amount of gas. The function iterates through the `featuredList` array, which contains all featured `AggregateToken` contracts, until it locates the target contract to be unfeatured.

>_ *nest/src/NestStaking.sol* SOLIDITY

```
function unfeatureToken(
    IAggregateToken aggregateToken
) external onlyRole(ADMIN_ROLE) {
    NestStakingStorage storage $ = _getNestStakingStorage();
    if (!$.isFeatured[aggregateToken]) {
        revert TokenNotFeatured(aggregateToken);
    }
    IAggregateToken[] storage featuredList = $.featuredList;
    uint256 length = featuredList.length;
    for (uint256 i = 0; i < length; ++i) {
        if (featuredList[i] == aggregateToken) {
            featuredList[i] = featuredList[length - 1];
            featuredList.pop();
            break;
        }
    }
    $.isFeatured[aggregateToken] = false;
    emit TokenUnfeatured(aggregateToken);
}
```

For each iteration, the function performs a comparison and accesses an element in storage, incurring gas costs. Since there is no limit on the number of `AggregateToken` contracts that can be deployed by users via `createAggregateToken`, users can substantially increase the cost of calling `unfeatureToken` by repeatedly invoking `createAggregateToken`, causing more `AggregateToken` contracts to be created and stored in the `featuredList` array. As a result, `unfeatureToken` becomes prohibitively expensive for the admin, who may no longer be able to execute this function.

Remediation

Instead of utilizing an array to store featured `AggregateToken` contracts, use a mapping for constant-time lookups.

Patch

Resolved in 223af8d.

05 — General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-PNT-SUG-00	Suggestions to ensure adherence to coding best practices.
OS-PNT-SUG-01	`withdraw` rounds down when calculating shares, potentially allowing users to withdraw more assets than justified by the shares they have burned.
OS-PNT-SUG-02	`AggregateToken` risks liquidity shortfalls during redemptions if assets remain locked in `ComponentTokens` and there is no mechanism to handle failed asynchronous deposits or redemptions resulting in stuck funds.

Code Maturity

OS-PNT-SUG-00

Description

1. Modify `NestBoringVaultModule.requestRedeem` and `NestBoringVaultModule.deposit` to revert with `unimplemented`, as done in the other functions. Also, ensure that `addToWhitelist` and `removeFromWhitelist` revert when the whitelist is disabled in `YieldToken`.
2. Utilize `getRateInQuoteSafe` instead of `getRateInQuote` during conversions in `NestBoringVaultModule` to improve safety when handling exchange rates.

Remediation

Implement the above-mentioned suggestions.

Incorrect Rounding Direction in Withdraw Functionality OS-PNT-SUG-01

Description

In the current implementation of `AggregateToken::convertToShares`, the calculation uses rounding down when converting assets to shares. This may result in underestimating the number of shares required for the requested assets. Therefore, a user may effectively withdraw more assets than the shares they have redeemed justify, creating a discrepancy in the vault's accounting. Consequently, this could lead to a loss of funds, especially when dealing with large share values.

>_ nest/src/ComponentToken.sol SOLIDITY

```
function previewWithdraw(
    uint256 assets
) public view virtual override(ERC4626Upgradeable, IERC4626) returns (uint256 shares) {
    [...]
    shares = convertToShares(assets);
}
```

>_ nest/src/AggregateToken.sol SOLIDITY

```
function convertToShares(
    uint256 assets
) public view override(ComponentToken, IComponentToken) returns (uint256 shares) {
    return assets * _BASE / _getAggregateTokenStorage().askPrice;
}
```

Remediation

Utilize a rounding-up strategy in `convertToShares` when determining how many shares to burn. This ensures that users always burn at least as many shares as necessary to match the withdrawn assets.

Absence of Functionality to Handle Edge Cases OS-PNT-SUG-02

Description

In `AggregateToken`, it is possible that when shares are withdrawn or redeemed, sufficient assets are unavailable because they are still locked in `ComponentTokens`. The `AggregateToken` contract holds its underlying assets in `ComponentTokens`, which may not be readily available, resulting in depleted `AggregateToken` liquidity or failure to execute withdraw or redeem operations.

Also, `AggregateToken` operate within workflows where deposits and redemptions may occur asynchronously. However, currently there is no mechanism to refund or cancel failed async deposits or redemptions. As a result, users may face situations where their assets or funds are stuck indefinitely.

Remediation

Modify `AggregateToken` to handle the above-stated edge cases.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.