Plume

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

Plumenetwork engaged OtterSec to assess the `arc-tokenization-contracts` and `nest-staking-contracts` programs. 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 11 findings throughout this audit engagement.

In particular, we identified several high-risk vulnerabilities, including a yield mechanism that incorrectly inflates users' share values (OS-PLM-ADV-00), inconsistencies in ERC4626 overridden functions due to yield token's custom conversion logic (OS-PLM-ADV-01), and inheritance conflicts between ERC4626 and the yield distribution token (OS-PLM-ADV-02).

Another potential vulnerability lies 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-PLM-ADV-04). Additionally, asynchronous deposit and redeem operations fail to accurately adjust pre-stored claimable shares and assets, potentially allowing incorrect accounting through minting or withdrawing disproportionate values compared to the specified inputs (OS-PLM-ADV-05)

Furthermore, discrepancies in the assets-to-shares conversion rates during mint and withdraw operations result in reversions if the calculated values exceed the initially requested amounts (OS-PLM-ADV-06). Moreover, there is 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-PLM-ADV-08).

We also made suggestions to ensure adherence to coding best practices (OS-PLM-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.

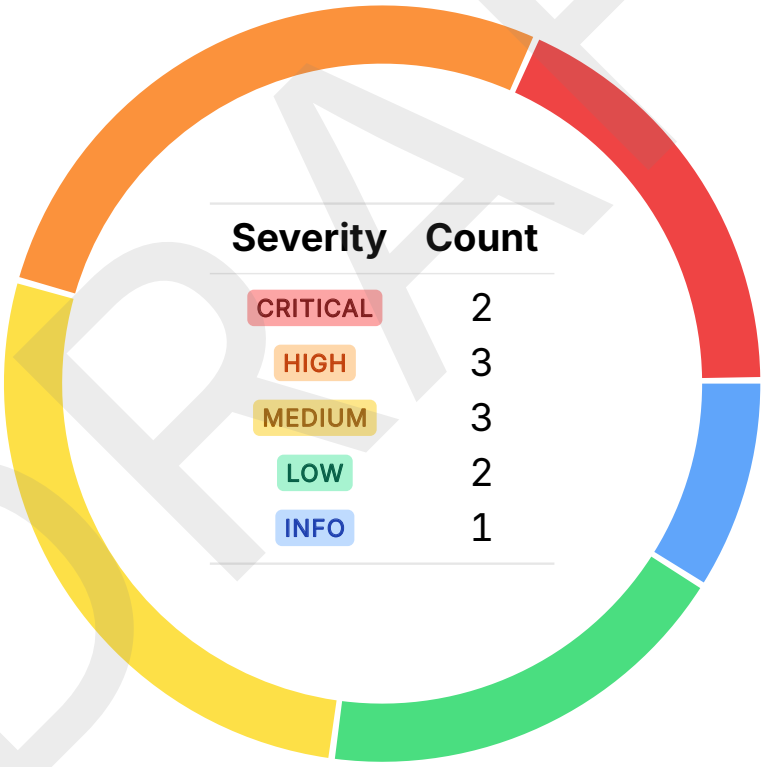
A brief description of the programs is as follows:

Name	Description
arc-tokenization-contracts	These contracts enable the creation of permissionless tokens that represent yield from real-world assets, allowing transactions on Plume to be executed directly on EOAs.
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 11 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	2
HIGH	3
MEDIUM	3
LOW	2
INFO	1

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-PLM-ADV-00	CRITICAL	RESOLVED	`receiveYield` incorrectly inflates share values when depositing yield rewards into the contract, as the increased token balance affects the share-to-token conversion ratio used for all users.
OS-PLM-ADV-01	CRITICAL	RESOLVED	`YieldToken`'s custom logic for `convertToShares` and `convertToAssets` creates inconsistencies with unmodified dependent methods in `ERC4626` which rely on these functions, resulting in potential mismatches in expected outcomes.
OS-PLM-ADV-02	HIGH	RESOLVED	The inheritance conflict prioritizes `ERC4626.decimals` function over `YieldDistributionToken.decimals`, resulting in mismatched token decimal logic and inconsistencies in token-related operations.
OS-PLM-ADV-03	HIGH	RESOLVED	`AggregateToken` lacks functionality to trigger `_notifyRedeem`, resulting in unprocessed asynchronous redemption requests despite the fact that the `asyncRedeem` feature is enabled by default.
OS-PLM-ADV-04	HIGH	RESOLVED	`buyVaultToken` risks misallocated approvals by granting access to the teller instead of the vault and relies on untrusted user-provided addresses, potentially allowing share inflation and asset recovery exploitation.

OS-PLM-ADV-05	MEDIUM	RESOLVED	`deposit` and `redeem` functions allow partial share redemptions against pre-stored asset values, leading to incorrect accounting during execution.
OS-PLM-ADV-06	MEDIUM	RESOLVED	`mint` and `withdraw` are vulnerable to discrepancies in the assets-to-shares conversion rate, which result in reversion if the calculated values during processing exceed the initially requested amounts.
OS-PLM-ADV-07	MEDIUM	RESOLVED	`unfeatureToken` in `NestStaking` has a linear gas cost increase due to the iteration over the `featuredList` to find and remove a token, resulting in high gas costs when the list grows substantially.
OS-PLM-ADV-08	LOW	TODO	`withdraw` rounds down when calculating shares, potentially allowing users to withdraw more assets than justified by the shares burned.
OS-PLM-ADV-09	LOW	TODO	`AggregateToken` risks liquidity shortfalls during redemptions if assets remain locked in `ComponentTokens` and there is no mechanism to handle failed asynchronous deposits or redemptions in `YieldToken` and `AggregateToken`, resulting in stuck funds.

Yield Distribution Share Inflation CRITICAL OS-PLM-ADV-00

Description

The issue concerns how `receiveYield` in `YieldToken` interacts with the contract's accounting, specifically the mechanism used to track user share values relative to the underlying assets. In the current implementation, when `receiveYield` is called, it increases the stored `yieldPerTokenStored` and the total `currencyToken` held by the `YieldToken` contract.

>_ *smart-wallets/src/token/YieldToken.sol* SOLIDITY

```
function receiveYield(IAssetToken assetToken, IERC20 currencyToken, uint256 currencyTokenAmount)
    ↪  external {
    [...]
    _depositYield(currencyTokenAmount);
}
```

>_ *smart-wallets/src/token/YieldDistributionToken.sol* SOLIDITY

```
function _depositYield(
    uint256 currencyTokenAmount
) internal {
    [...]
    if (currentSupply > 0) {
        [...]
        $.yieldPerTokenStored += currencyTokenAmount.mulDiv(SCALE, divisor);
    }
    [...]
    $.currencyToken.safeTransferFrom(_msgSender(), address(this), currencyTokenAmount);
    emit Deposited(_msgSender(), currencyTokenAmount);
}
```

However, there is an oversight in this mechanism. When depositing `currencyTokenAmount` as yield, the `totalAssets` held by the contract increases. This creates a dual effect where users not only receive their intended yield but also experience an unintended share value inflation due to the share-to-asset conversion dependency. As a result, users receive more value than they should be entitled to.

Remediation

Separate the yield distribution process to prevent direct inflation of share value.

Patch

Resolved in 223af8d.

Inconsistent Function Override Logic CRITICAL OS-PLM-ADV-01

Description

The vulnerability concerns inconsistencies that arise when certain functions in the `ERC4626` implementation define custom logic without overriding the dependent functions in the base `ERC4626` contract to reflect this custom logic. `YieldToken` redefines `convertToShares` and `convertToAssets` with custom logic that differs from the inherited `ERC4626` contract's expectations. These methods directly impact proportionality calculations between assets and shares.

>_ *smart-wallets/src/token/YieldToken.sol* SOLIDITY

```solidity
function convertToShares(
    uint256 assets
) public view override(ERC4626, IComponentToken) returns (uint256 shares) {
    uint256 supply = totalSupply();
    uint256 totalAssets_ = totalAssets();
    if (supply == 0 || totalAssets_ == 0) {
        return assets;
    }
    return (assets * supply) / totalAssets_;
}

function convertToAssets(
    uint256 shares
) public view override(ERC4626, IComponentToken) returns (uint256 assets) {
    uint256 supply = totalSupply();
    if (supply == 0) {
        return shares;
    }
    return (shares * totalAssets()) / supply;
}
```

Functions such as `maxWithdraw`, `previewDeposit`, `previewMint`, `previewWithdraw`, and `previewRedeem` depend on `convertToShares` and `convertToAssets` for accurate calculations. However, the `YieldToken` implementation does not override these functions, resulting in them relying on the base `ERC4626` versions. Similarly, `deposit` and `mint` in `ERC4626` also depend indirectly on `convertToShares` or `convertToAssets`.

When users interact with `YieldToken` through inherited methods, these methods, which utilize the `ERC4626` logic that assumes the default behavior of `convertToShares` and `convertToAssets`, may yield incorrect results. This also applies to `ComponentToken::maxWithdraw` as well.

Remediation

Override all the dependent functions to align their behavior with the custom logic defined in `convertToShares` and `convertToAssets` to ensure consistency across all operations.

Patch

Resolved in 4f16028.

Inheritance Conflict in Decimals Method HIGH OS-PLM-ADV-02

Description

The vulnerability lies in how Solidity's inheritance hierarchy and the `super` keyword determine which parent implementation is prioritized when overriding a function. Here, `YieldToken` inherits both `YieldDistributionToken` and `ERC4626`, which both implement `decimals`.

>_ *smart-wallets/src/token/YieldToken.sol* SOLIDITY

```
contract YieldToken is YieldDistributionToken, ERC4626, WalletUtils, IYieldToken,
    ↪  IComponentToken {
    [...]
    /// @inheritdoc ERC20
    function decimals() public view override(YieldDistributionToken, ERC4626) returns (uint8) {
        return super.decimals();
    }
    [...]
}
```

Since `super` prioritizes the parent contract that appears last in the inheritance chain, the definition of `decimals` in `ERC4626` will take precedence over the one in `YieldDistributionToken`. As a result, the logic intended by `YieldDistributionToken` for `decimals` may be ignored, resulting in returning incorrect decimals. `YieldDistributionToken.decimals` is defined to always return 8, while `ERC4626.decimals` dynamically calculates the value based on the underlying asset.

Remediation

Avoid relying on the `super` resolution. `YieldToken` should explicitly define its desired behavior.

Patch

Resolved in 4f16028.

Unprocessed Async Redemption Requests HIGH OS-PLM-ADV-03

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 makes the redemption requests remain in the `pendingRedeemRequest` mapping indefinitely and the `claimableRedeemRequest` mapping is never updated. Thus, it will not be possible for users to claim their assets or redeemed tokens.

Remediation

Ensure that `AggregateToken` calls `_notifyRedeem` to close the asynchronous redemption life-cycle.

Patch

Resolved in dbacef9.

Lack of Proper Access Control Logic in BuyVaultToken HIGH OS-PLM-ADV-04

Description

There are multiple vulnerabilities that arise from `buyVaultToken` in `AggregateToken` as a result of inconsistent access control logic. The function currently 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. A user may 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 the approval is be granted to the `vault` instead of the `teller` in `buyVaultToken`, and refrain from trusting user-provided address inputs. Also, limit access to `buyVaultToken` such that only the `MANAGER_ROLE` may access it as enforced in `buyComponentToken`.

Patch

Resolved in 223af8d.

Proportionality Violation in Deposit and Redeem MEDIUM OS-PLM-ADV-05

Description

In `YieldToken` the expected logic of proportionality between assets and shares is violated. In `deposit`, regardless of the assets that is passed by the user, the number of shares minted is the maximum number stored in the `$.sharesDepositRequest[controller]` mapping. The value in `$.sharesDepositRequest` does not dynamically compute proportionality based on assets. Instead, it is treated as a static, pre-set value.

Similarly, in `redeem`, regardless of the shares passed by the user, the assets transferred are determined by the maximum stored in `$.assetsRedeemRequest[controller]`, not calculated based on the proportionality of shares. Thus, share ownership no longer reflects the actual amount of underlying assets deposited, and users will be unable to trust the vault to fairly distribute assets based on ownership.

Remediation

Ensures that the number of minted shares reflects the proportion of assets deposited relative to the vault's current total assets and shares, and the number of assets redeemed reflects the proportion of shares burned relative to total vault shares.

Patch

Resolved in 4f16028.

Conversion Rate Discrepancy MEDIUM OS-PLM-ADV-06

Description

A discrepancy between assets and shares may occur due to potential changes in the conversion rate between the request and processing times during `YieldToken::mint` or `YieldToken::withdraw`. This may result in the functions to revert if the conversion result exceeds the requested value. Thus, valid mint or withdrawal requests may be rejected simply due to conversion rate changes.

Remediation

Record the `convertToAssets` or `convertToShares` value at request time and utilize that fixed rate during `mint` or `withdraw` execution.

Patch

Resolved in 4f16028.

Excessive Gas Cost for Execution of UnfeatureToken MEDIUM OS-PLM-ADV-07

Description

The issue demonstrates how users can manipulate `NestStaking::unfeatureToken` to consume excessive 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 the element from storage, which incurs gas costs. Since there is no limit on the number of aggregate tokens that can be deployed by users via `createAggregateToken`, users can make the cost of calling `unfeatureToken` increase substantially by repeatedly calling `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 the list of featured `AggregateToken` contracts, utilize a mapping for constant time lookups.

Patch

Resolved in 223af8d.

Incorrect Rounding Direction in Withdraw Functionality LOW OS-PLM-ADV-08

Description

In the current implementation of `AggregateToken::convertToShares`, the calculation utilizes rounding down when converting assets to shares. This may result in underestimating the number of shares required for the requested assets. Thus, a user may effectively withdraw more assets than they have redeemed shares for, creating a discrepancy in the vault's accounting. Consequently, this will result in 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;
}
```

This also applies to `YieldToken::convertToShares`.

>_ *smart-wallets/src/token/YieldToken.sol* SOLIDITY

```
function convertToShares(
    uint256 assets
) public view override(ERC4626, IComponentToken) returns (uint256 shares) {
    [...]
    return (assets * supply) / totalAssets_;
}
```

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 LOW OS-PLM-ADV-09

Description

Both `AggregateToken` and `YieldToken` fail to handle specific edge cases impacting the optimal functioning of these programs. In `AggregateToken` it is possible that when shares are withdrawn or redeemed, sufficient assets are unavailable because they are still locked in `ComponentTokens`. `AggregateToken` contract holds its underlying assets in `ComponentTokens`, which may not be readily available, resulting in depleting `AggregateToken` liquidity or failure to execute a withdraw or redeem operation.

Also, `YieldToken` and `AggregateToken` operate within workflows where deposits and redemptions may occur asynchronously. Howwever, 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` and `YieldToken` to handle the above stated edge cases.

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-PLM-SUG-00	Suggestions to ensure adherence to coding best practices.

Code Maturity OS-PLM-SUG-00

Description

1. Modify `NestBoringVaultModule.requestRedeem` and `NestBoringVaultModule.deposit` such that revert with `unimplemented` as done in the other functions. Also, ensure `addToWhitelist` and `removeFromWhitelist` revert when the whitelist is disabled in `YieldToken`.
2. Refactor `sellVaultToken` to utilize `safeUpdateAtomicRequest` instead of `updateAtomicRequest` to ensure improved safety during the process of submitting an atomic request.

>_ smart-wallets/src/token/YieldToken.sol SOLIDITY

```
function sellVaultToken(
    [...]
) public nonReentrant onlyRole(MANAGER_ROLE) returns (uint256) {
    [...]
    IAtomicQueue queue = IAtomicQueue(_atomicQueue);
    queue.updateAtomicRequest(IERC20(offerToken), IERC20(wantToken), request);
    emit VaultTokenSellRequested(msg.sender, offerToken, shares, price);
    return REQUEST_ID;
}
```

3. Utilize `getRateInQuoteSafe` instead of `getRateInQuote` during conversions in `NestBoringVaultModule` to improving safety when handling exchange rates.

Remediation

Implement the above-mentioned suggestions.

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.