Smart Contract Security Audit Report

Table Of Contents

1 Executive Summary

On 2024.12.03, the SlowMist security team received the Plume Network team's security audit application for Nest Contracts, developed the audit plan according to the agreement of both parties and the characteristics of the project, and finally issued the security audit report.

The SlowMist security team adopts the strategy of "white box lead, black, grey box assists" to conduct a complete security test on the project in the way closest to the real attack.

The test method information:

Test method	Description
Black box testing	Conduct security tests from an attacker's perspective externally.
Grey box testing	Conduct security testing on code modules through the scripting tool, observing the internal running status, mining weaknesses.
White box testing	Based on the open source code, non-open source code, to detect whether there are vulnerabilities in programs such as nodes, SDK, etc.

The vulnerability severity level information:

Level	Description
Critical	Critical severity vulnerabilities will have a significant impact on the security of the DeFi project, and it is strongly recommended to fix the critical vulnerabilities.
High	High severity vulnerabilities will affect the normal operation of the DeFi project. It is strongly recommended to fix high-risk vulnerabilities.
Medium	Medium severity vulnerability will affect the operation of the DeFi project. It is recommended to fix medium-risk vulnerabilities.
Low	Low severity vulnerabilities may affect the operation of the DeFi project in certain scenarios. It is suggested that the project team should evaluate and consider whether these vulnerabilities need to be fixed.
Weakness	There are safety risks theoretically, but it is extremely difficult to reproduce in engineering.
Suggestion	There are better practices for coding or architecture.

2 Audit Methodology

The security audit process of SlowMist security team for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using automated analysis tools.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that was considered during the audit of the smart contract:

Serial Number	Audit Class	Audit Subclass
1	Overflow Audit	-
2	Reentrancy Attack Audit	-
3	Replay Attack Audit	-
4	Flashloan Attack Audit	-
5	Race Conditions Audit	Reordering Attack Audit
6	Permission Vulnerability Audit	Access Control Audit
		Excessive Authority Audit
7	Security Design Audit	External Module Safe Use Audit
		Compiler Version Security Audit
		Hard-coded Address Security Audit
		Fallback Function Safe Use Audit
		Show Coding Security Audit
		Function Return Value Security Audit
		External Call Function Security Audit

Serial Number	Audit Class	Audit Subclass
7	Security Design Audit	Block data Dependence Security Audit
		tx.origin Authentication Security Audit
8	Denial of Service Audit	-
9	Gas Optimization Audit	-
10	Design Logic Audit	-
11	Variable Coverage Vulnerability Audit	-
12	"False Top-up" Vulnerability Audit	-
13	Scoping and Declarations Audit	-
14	Malicious Event Log Audit	-
15	Arithmetic Accuracy Deviation Audit	-
16	Uninitialized Storage Pointer Audit	-

3 Project Overview

3.1 Project Introduction

This is the Nest Stake product from Plume Network. Users can permissionlessly deposit and withdraw funds from Nest Stake, where they can deposit stablecoins like $USDC/$USDT to convert them into stake shares.

3.2 Vulnerability Information

The following is the status of the vulnerabilities found in this audit:

NO	Title	Category	Level	Status
N1	Risks of Token Compatibility	Design Logic Audit	Suggestion	Fixed

NO	Title	Category	Level	Status
N2	Asynchronous deposit/redemption design flaw	Design Logic Audit	Critical	Fixed
N3	Wrong asset check when async minting	Design Logic Audit	Medium	Fixed
N4	Risks of excessive privilege	Authority Control Vulnerability Audit	Medium	Acknowledged
N5	ERC4626 preview function override is incorrect	Design Logic Audit	Critical	Fixed
N6	AggregateToken contract paused status check coverage is incomplete	Design Logic Audit	Medium	Fixed
N7	The approveComponentToken operation does not check the validity of the componentToken	Design Logic Audit	Low	Fixed
N8	Unable to remove componentToken	Others	Suggestion	Fixed
N9	Users may not be able to withdraw funds from the AggregateToken contract in real time	Others	Information	Acknowledged
N10	Input parameters lack zero-address validation during initialization	Others	Suggestion	Fixed
N11	Redundant input parameters	Others	Suggestion	Acknowledged
N12	Potential risks of using different participation rates	Design Logic Audit	Medium	Fixed
N13	Invalid balance acquisition	Design Logic Audit	Low	Fixed

4 Code Overview

4.1 Contracts Description

Audit Version:

https://github.com/plumenetwork/contracts

commit: 8711e8409cd7a1a32c10e65dbb0382286a9ff9e3

Fixed Version:

https://github.com/plumenetwork/contracts

commit: 1b4ab430e1fddf2161fe6423d442e9a166529eed

Audit Scope:

```
./nest/src/
├── AggregateToken.sol
├── ComponentToken.sol
└── token
    ├── BoringVaultAdapter.sol
    └── pUSD.sol
```

The main network address of the contract is as follows:

The code was not deployed to the mainnet.

4.2 Visibility Description

The SlowMist Security team analyzed the visibility of major contracts during the audit, the result as follows:

AggregateToken			
Function Name	Visibility	Mutability	Modifiers
_getAggregateTokenStorage	Private	-	-
<Constructor>	Public	Can Modify State	-
initialize	Public	Can Modify State	initializer
convertToShares	Public	-	-
convertToAssets	Public	-	-

AggregateToken			
asset	Public	-	-
deposit	Public	Can Modify State	-
redeem	Public	Can Modify State	-
totalAssets	Public	-	-
approveComponentToken	External	Can Modify State	nonReentrant onlyRole
addComponentToken	External	Can Modify State	nonReentrant onlyRole
buyComponentToken	Public	Can Modify State	nonReentrant onlyRole
sellComponentToken	Public	Can Modify State	nonReentrant onlyRole
requestBuyComponentToken	Public	Can Modify State	nonReentrant onlyRole
requestSellComponentToken	Public	Can Modify State	nonReentrant onlyRole
setAskPrice	External	Can Modify State	nonReentrant onlyRole
setBidPrice	External	Can Modify State	nonReentrant onlyRole
pause	External	Can Modify State	onlyRole
unpause	External	Can Modify State	nonReentrant onlyRole
getAskPrice	External	-	-
getBidPrice	External	-	-
getComponentTokenList	Public	-	-
isPaused	External	-	-
getComponentToken	Public	-	-
supportsInterface	Public	-	-

ComponentToken			
Function Name	Visibility	Mutability	Modifiers

ComponentToken			
_getComponentTokenStorage	Internal	-	-
<Constructor>	Public	Can Modify State	-
initialize	Public	Can Modify State	onlyInitializing
_authorizeUpgrade	Internal	Can Modify State	onlyRole
supportsInterface	Public	-	-
asset	Public	-	-
totalAssets	Public	-	-
assetsOf	Public	-	-
convertToShares	Public	-	-
convertToAssets	Public	-	-
requestDeposit	Public	Can Modify State	nonReentrant
_notifyDeposit	Internal	Can Modify State	nonReentrant
deposit	Public	Can Modify State	nonReentrant
mint	Public	Can Modify State	nonReentrant
requestRedeem	Public	Can Modify State	nonReentrant
_notifyRedeem	Internal	Can Modify State	nonReentrant
redeem	Public	Can Modify State	nonReentrant
withdraw	Public	Can Modify State	nonReentrant
share	External	-	-
isOperator	Public	-	-
pendingDepositRequest	Public	-	-
claimableDepositRequest	Public	-	-

ComponentToken			
pendingRedeemRequest	Public	-	-
claimableRedeemRequest	Public	-	-
previewDeposit	Public	-	-
previewMint	Public	-	-
previewRedeem	Public	-	-
previewWithdraw	Public	-	-
setOperator	Public	-	-

BoringVaultAdapter			
Function Name	Visibility	Mutability	Modifiers
_getBoringVaultAdapterStorage	Private	-	-
<Constructor>	Public	Can Modify State	-
initialize	Public	Can Modify State	onlyInitializing
reinitialize	Public	Can Modify State	onlyRole
_authorizeUpgrade	Internal	Can Modify State	onlyRole
getVault	External	-	-
getTeller	External	-	-
getAtomicQueue	External	-	-
version	Public	-	-
deposit	Public	Can Modify State	-
requestRedeem	Public	Can Modify State	-
notifyRedeem	External	Can Modify State	onlyRole
redeem	Public	Can Modify State	-

BoringVaultAdapter			
previewDeposit	Public	-	-
previewRedeem	Public	-	-
convertToShares	Public	-	-
convertToAssets	Public	-	-
transfer	Public	Can Modify State	-
transferFrom	Public	Can Modify State	-
balanceOf	Public	-	-
assetsOf	Public	-	-
decimals	Public	-	-
supportsInterface	Public	-	-

pUSD			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
initialize	Public	Can Modify State	initializer
name	Public	-	-
symbol	Public	-	-

4.3 Vulnerability Summary

[N1] [Suggestion] Risks of Token Compatibility

Category: Design Logic Audit

Content

In the ComponentToken contract, users can perform deposit, redemption, and withdrawal operations through the requestDeposit/deposit/mint/redeem/withdraw functions. During these operations, the contract initiates token

transfers by calling the underlying assets supported by the ComponentToken contract. The contract executes token transfers through transfer or transferFrom functions and verifies their return values. However, it's important to note that some tokens (such as USDT on the mainnet) do not fully comply with the ERC20 standard and may return false even when transfers are successful. This can prevent the protocol from properly handling deposits and withdrawals of these tokens.

Code location: nest/src/ComponentToken.sol#L256,L313,L347,L434,L467

```
    function requestDeposit(
        uint256 assets,
        address controller,
        address owner
    ) public virtual nonReentrant returns (uint256 requestId) {
        ...
        if (!IERC20(asset()).transferFrom(owner, address(this), assets)) {
            revert InsufficientBalance(IERC20(asset()), owner, assets);
        }
        ...
    }

    function deposit(
        uint256 assets,
        address receiver,
        address controller
    ) public virtual nonReentrant returns (uint256 shares) {
        ...
            if (!IERC20(asset()).transferFrom(controller, address(this), assets)) {
                revert InsufficientBalance(IERC20(asset()), controller, assets);
            }
        ...
    }

    function redeem(
        uint256 shares,
        address receiver,
        address controller
    ) public virtual override(ERC4626Upgradeable, IERC7540) nonReentrant returns
(uint256 assets) {
        ...
        if (!IERC20(asset()).transfer(receiver, assets)) {
            revert InsufficientBalance(IERC20(asset()), address(this), assets);
        }
        ...
    }
```

```
    function withdraw(
        uint256 assets,
        address receiver,
        address controller
    ) public virtual override(ERC4626Upgradeable, IERC7540) nonReentrant returns
(uint256 shares) {
        ...
        if (!IERC20(asset()).transfer(receiver, assets)) {
            revert InsufficientBalance(IERC20(asset()), address(this), assets);
        }
        ...
    }
```

Solution

Although the stablecoins on the Plume network have not encountered this issue yet, it is recommended to implement the OpenZeppelin SafeERC20 library for token transfers to mitigate these potential risks.

Status

Fixed; Fixed in PR 121

[N2] [Critical] Asynchronous deposit/redemption design flaw

Category: Design Logic Audit

Content

In the ComponentToken contract, users can initiate asynchronous deposits through the requestDeposit function, and the contract uses the `_notifyDeposit` function to set the allowable amount of assets (claimableDepositRequest) and corresponding shares (sharesDepositRequest) for asynchronous deposits. Once the asynchronous deposit is approved, users can complete the deposit through the deposit function, which deducts the asynchronous deposit amount (claimableDepositRequest) and all corresponding shares (sharesDepositRequest) from the user's account. Unfortunately, when users execute deposits through the deposit function, the input `assets` amount may be significantly lower than the contract's recorded `claimableDepositRequest`. This results in only a partial deposit being processed while all asynchronous shares are deducted, leading to accounting inconsistencies in the asynchronous data.

Similarly, this vulnerability exists in the redeem function as well.

Code location:

nest/src/ComponentToken.sol#L309-L311

```
    function deposit(
        uint256 assets,
        ...
    ) public virtual nonReentrant returns (uint256 shares) {
        ...
        if ($.asyncDeposit) {
            if ($.claimableDepositRequest[controller] < assets) {
                revert InsufficientRequestBalance(controller, assets, 1);
            }
            shares = $.sharesDepositRequest[controller];
            $.claimableDepositRequest[controller] -= assets;
            $.sharesDepositRequest[controller] -= shares;
        } ...
    }
```

nest/src/ComponentToken.sol#L426-L428

```
    function redeem(
        uint256 shares,
        ...
    ) public virtual override(ERC4626Upgradeable, IERC7540) nonReentrant returns
(uint256 assets) {
        ...
        if ($.asyncRedeem) {
            if ($.claimableRedeemRequest[controller] < shares) {
                revert InsufficientRequestBalance(controller, shares, 3);
            }
            assets = $.assetsRedeemRequest[controller];
            $.claimableRedeemRequest[controller] -= shares;
            $.assetsRedeemRequest[controller] -= assets;
        } ...
    }
```

Solution

It is recommended to eliminate user-specified amounts during asynchronous deposits/redemptions. Instead, the contract should process transactions using the pre-recorded asynchronous deposit/withdrawal amounts. Specifically, when executing deposits/redemptions, all recorded request amounts should be reset to 0.

Status

Fixed; Fixed in PR 119

[N3] [Medium] Wrong asset check when async minting

Category: Design Logic Audit

Content

In the ComponentToken contract, when users execute asynchronous minting through the mint function, the contract verifies that the asset amount corresponding to the shares being minted is less than or equal to `claimableDepositRequest`. However, it's important to note that the `claimableDepositRequest` amount was previously set by the `_notifyDeposit` function. This means that during the mint operation, the calculated assets amount may differ from the amount recorded during the `_notifyDeposit` operation due to potential price fluctuations. Therefore, using real-time `convertToAssets` conversion to check against the previously recorded `claimableDepositRequest` is not logically sound.

The same is true for the withdraw function.

Code location:

nest/src/ComponentToken.sol#L341

```
    function mint(
        uint256 shares,
        address receiver,
        address controller
    ) public virtual nonReentrant returns (uint256 assets) {
        ...
        if ($.asyncDeposit) {
            if ($.claimableDepositRequest[controller] < assets) {
                revert InsufficientRequestBalance(controller, assets, 1);
            }
            $.claimableDepositRequest[controller] -= assets;
            $.sharesDepositRequest[controller] -= shares;
        } ...
    }
```

nest/src/ComponentToken.sol#L458

```
    function withdraw(
        uint256 assets,
        address receiver,
        address controller
    ) public virtual override(ERC4626Upgradeable, IERC7540) nonReentrant returns
(uint256 shares) {
        ...
        shares = convertToShares(assets);
```

```
        if ($.asyncRedeem) {
            if ($.claimableRedeemRequest[controller] < shares) {
                revert InsufficientRequestBalance(controller, shares, 3);
            }
            $.claimableRedeemRequest[controller] -= shares;
            $.assetsRedeemRequest[controller] -= assets;
        } ...
    }
```

Solution

It is recommended to perform checks using `sharesDepositRequest` directly, eliminating the need for `convertToAssets` operation and asset comparison. However, as mentioned in N2, using the entire `claimableDepositRequest` and `sharesDepositRequest` as deposit amounts during asynchronous minting would prevent accounting inconsistencies.

Status

Fixed; Fixed in PR 122

[N4] [Medium] Risks of excessive privilege

Category: Authority Control Vulnerability Audit

Content

In the BoringVaultAdapter contract, the admin role can set the allowable amount for users' asynchronous redemptions through the notifyRedeem function. It's concerning that these amounts are directly input by the admin, and there is no correlation between the input assets and shares. This creates a risk of excessive privilege for the admin role.

In the AggregateToken contract, the admin role can add any address to the componentTokenList and can approve the contract's `componentToken.asset()` tokens to any componentToken contract through the approveComponentToken function. This creates a risk of excessive admin privileges. Additionally, the PRICE_UPDATER_ROLE can arbitrarily modify prices through setAskPrice and setBidPrice functions, which directly affect user assets/shares conversion operations, leading to potential excessive privilege risks for the PRICE_UPDATER_ROLE.

In the BoringVaultAdapter contract, the UPGRADER_ROLE can repeatedly initialize and arbitrarily modify critical contract parameters through the reinitialize function. This creates a risk of excessive privileges for the UPGRADER_ROLE.

Code location:

nest/src/token/BoringVaultAdapter.sol#L317

```
    function notifyRedeem(uint256 assets, uint256 shares, address controller)
external onlyRole(ADMIN_ROLE) {
        _notifyRedeem(assets, shares, controller);
    }
```

nest/src/ComponentToken.sol#L403

```
    function _notifyRedeem(uint256 assets, uint256 shares, address controller)
internal virtual nonReentrant {
        ...

        $.pendingRedeemRequest[controller] -= shares;
        $.claimableRedeemRequest[controller] += shares;
        $.assetsRedeemRequest[controller] += assets;

        emit RedeemNotified(controller, assets, shares);
    }
```

AggregateToken.sol#L200-L222,L299-L314

```
    function approveComponentToken(
        IComponentToken componentToken,
        uint256 amount
    ) external nonReentrant onlyRole(ADMIN_ROLE) {
        IERC20(componentToken.asset()).approve(address(componentToken), amount);
    }

    function addComponentToken(
        IComponentToken componentToken
    ) external nonReentrant onlyRole(ADMIN_ROLE) {
        AggregateTokenStorage storage $ = _getAggregateTokenStorage();
        if ($.componentTokenMap[componentToken]) {
            revert ComponentTokenAlreadyListed(componentToken);
        }
        $.componentTokenList.push(componentToken);
```

```
        $.componentTokenMap[componentToken] = true;
        emit ComponentTokenListed(componentToken);
    }

    function setAskPrice(
        uint256 askPrice
    ) external nonReentrant onlyRole(PRICE_UPDATER_ROLE) {
        _getAggregateTokenStorage().askPrice = askPrice;
    }

    function setBidPrice(
        uint256 bidPrice
    ) external nonReentrant onlyRole(PRICE_UPDATER_ROLE) {
        _getAggregateTokenStorage().bidPrice = bidPrice;
    }
```

nest/src/token/BoringVaultAdapter.sol#L143-L171

```
    function reinitialize(
        address owner,
        IERC20 asset_,
        address vault_,
        address teller_,
        address atomicQueue_,
        address lens_,
        address accountant_
    ) public onlyRole(UPGRADER_ROLE) {
        ...
        // Increment version
        $.version += 1;
        $.boringVault.teller = ITeller(teller_);
        $.boringVault.vault = IVault(vault_);
        $.boringVault.atomicQueue = IAtomicQueue(atomicQueue_);
        $.boringVault.lens = ILens(lens_);
        $.boringVault.accountant = IAccountantWithRateProviders(accountant_);

        emit Reinitialized($.version);
    }
```

Solution

For the notifyRedeem operation, it is recommended to calculate the redeemable assets amount within the contract using convertToAssets based on the current assets-to-shares ratio, rather than allowing arbitrary setting of `assetsRedeemRequest`. This would prevent the risk of excessive privilege that comes with arbitrary value

assignment.

Regarding the privilege risks in the AggregateToken contract, in the short term, transferring privileged roles to multi-signature management can effectively mitigate single-point risks. In the long term, transitioning privileged roles to DAO governance can effectively resolve excessive privilege risks. During the transition period, implementing multi-signature management combined with timelock-delayed transaction execution can effectively mitigate the risks of excessive privileges.

For the reinitialize function in the BoringVaultAdapter contract, if this is not an intended design, it is recommended to use the reinitializer modifier to limit the number of initialization attempts to mitigate this risk.

Status

Acknowledged; After communicating with the project team, the project team stated that the admin role will be a multisig contract controlled by trusted third parties.

[N5] [Critical] ERC4626 preview function override is incorrect

Category: Design Logic Audit

Content

In the ComponentToken contract, which inherits from the ERC4626 contract, the convertToShares and convertToAssets functions have been reimplemented. While the contract overrides the preview functions accordingly, the previewMint and previewWithdraw functions still utilize the ERC4626 contract's implementation through `super.previewDeposit` / `super.previewWithdraw`. The ERC4626 contract's implementations of previewMint and previewWithdraw internally call `_convertToShares` and `_convertToAssets` functions, which haven't been reimplemented in the ComponentToken contract. This means these functions use conversion logic that differs from the ComponentToken contract. Consequently, users performing mint and withdraw operations through ERC4626 are inadvertently using inconsistent conversion logic, which deviates from the intended behavior.

Code location:

nest/src/ComponentToken.sol#L530,L557

```
    function previewMint(
        uint256 shares
    ) public view virtual override(ERC4626Upgradeable, IERC4626) returns (uint256
assets) {
        if (_getComponentTokenStorage().asyncDeposit) {
```

```
            revert Unimplemented();
        }
        assets = super.previewDeposit(shares);
    }

    function previewWithdraw(
        uint256 assets
    ) public view virtual override(ERC4626Upgradeable, IERC4626) returns (uint256
shares) {
        if (_getComponentTokenStorage().asyncRedeem) {
            revert Unimplemented();
        }
        shares = super.previewWithdraw(assets);
    }
```

ERC4626.sol#L157,L162

```
    function previewMint(uint256 shares) public view virtual returns (uint256) {
        return _convertToAssets(shares, Math.Rounding.Ceil);
    }

    function previewWithdraw(uint256 assets) public view virtual returns (uint256) {
        return _convertToShares(assets, Math.Rounding.Ceil);
    }
```

Solution

It is recommended to directly use the reimplemented convertToAssets and convertToShares functions when overriding the previewMint and previewWithdraw functions.

Status

Fixed; Fixed in PR 114

[N6] [Medium] AggregateToken contract paused status check coverage is incomplete

Category: Design Logic Audit

Content

In the AggregateToken contract, the deposit function overrides ComponentToken's deposit function to check if the contract is in a paused state. However, it's important to note that the AggregateToken contract does not override either the deposit and mint functions from the ERC4626 contract or the mint function from the ComponentToken contract. This oversight means that even when the AggregateToken contract is paused, users can still make deposits

through the ERC4626 contract's deposit and mint functions, as well as through the ComponentToken contract's mint function.

Code location: nest/src/AggregateToken.sol#L172

```
function deposit(
    uint256 assets,
    address receiver,
    address controller
) public override(ComponentToken, IComponentToken) returns (uint256 shares) {
    if (_getAggregateTokenStorage().paused) {
        revert DepositPaused();
    }
    return super.deposit(assets, receiver, controller);
}
```

Solution

It is recommended to implement pause state verification across all deposit-related functions.

Status

Fixed; Fixed in PR 112

[N7] [Low] The approveComponentToken operation does not check the validity of the componentToken

Category: Design Logic Audit

Content

In the AggregateToken contract, the admin role can use the approveComponentToken function to approve the contract's `componentToken.asset()` tokens to a specified componentToken. However, there is no verification to check whether this componentToken exists in the componentTokenMap list.

Code location: nest/src/AggregateToken.sol#L200-L205

```
function approveComponentToken(
    IComponentToken componentToken,
    uint256 amount
) external nonReentrant onlyRole(ADMIN_ROLE) {
    IERC20(componentToken.asset()).approve(address(componentToken), amount);
}
```

Solution

It is recommended to verify that the componentToken is present in the contract's componentTokenMap list before executing the approveComponentToken operation.

Status

Fixed; Fixed in PR 111

[N8] [Suggestion] Unable to remove componentToken

Category: Others

Content

In the AggregateToken contract, the admin role can add componentToken contracts to the componentTokenList and set their componentTokenMap status to true using the addComponentToken function. However, the contract lacks functionality to remove componentToken contracts from the componentTokenList. This makes it impossible to remove deprecated componentToken contracts when needed.

Code location: nest/src/AggregateToken.sol#L212

```
    function addComponentToken(
        IComponentToken componentToken
    ) external nonReentrant onlyRole(ADMIN_ROLE) {
        ...
    }
```

Solution

If this is not an intended design choice, it is recommended to implement functionality for removing componentToken contracts to avoid the aforementioned limitation.

Status

Fixed; Fixed in PR 118

[N9] [Information] Users may not be able to withdraw funds from the AggregateToken contract in real time

Category: Others

Content

In the AggregateToken contract, the admin role can deposit users' funds into specified componentToken contracts through the buyComponentToken function. Since users hold shares of the AggregateToken contract, when all assets in the AggregateToken contract are deposited into componentToken contracts, users cannot withdraw their funds independently. They must wait for the admin role to execute the sellComponentToken operation to retrieve their deposits.

Code location: nest/src/AggregateToken.sol#L231-L260

```
    function buyComponentToken(
        IComponentToken componentToken,
        uint256 assets
    ) public nonReentrant onlyRole(ADMIN_ROLE) {
        ...

        uint256 componentTokenAmount = componentToken.deposit(assets, address(this),
address(this));
        emit ComponentTokenBought(msg.sender, componentToken, componentTokenAmount,
assets);
    }

    function sellComponentToken(
        IComponentToken componentToken,
        uint256 componentTokenAmount
    ) public nonReentrant onlyRole(ADMIN_ROLE) {
        uint256 assets = componentToken.redeem(componentTokenAmount, address(this),
address(this));
        emit ComponentTokenSold(msg.sender, componentToken, componentTokenAmount,
assets);
    }
```

Solution

N/A

Status

Acknowledged

[N10] [Suggestion] Input parameters lack zero-address validation during initialization

Category: Others

Content

In the BoringVaultAdapter contract, both initialize and reinitialize functions lack zero-address validation for the input

parameters `lens_` and `accountant_`. These addresses are associated with the contract's assets/shares conversion functionality, and if set to zero addresses, certain features would become inoperable.

Code location: nest/src/token/BoringVaultAdapter.sol#L116-L121,L153-L158

```
    function initialize(
        ...
        address lens_,
        address accountant_,
        ...
    ) public onlyInitializing {
        if (
            owner == address(0) || address(asset_) == address(0) || vault_ ==
address(0) || teller_ == address(0)
                || atomicQueue_ == address(0)
        ) {
            revert ZeroAddress();
        }
        ...
    }

    function reinitialize(
        ...
        address lens_,
        address accountant_
    ) public onlyRole(UPGRADER_ROLE) {
        // Reinitialize as needed
        if (
            owner == address(0) || address(asset_) == address(0) || vault_ ==
address(0) || teller_ == address(0)
                || atomicQueue_ == address(0)
        ) {
            revert ZeroAddress();
        }
        ...
    }
```

Solution

It is recommended to implement zero-address validation for `lens_` and `accountant_` parameters during initialization.

Status

Fixed; Fixed in PR 110

[N11] [Suggestion] Redundant input parameters

Category: Others

Content

In the BoringVaultAdapter contract, the deposit function is used to deposit user funds into the vault contract through the teller contract. It's worth noting that although the function accepts a `controller` parameter, this input parameter is not used within the function, making it redundant.

Code location: nest/src/token/BoringVaultAdapter.sol#L232

```
function deposit(
    uint256 assets,
    address receiver,
    address controller,
    uint256 minimumMint
) public virtual returns (uint256 shares) {
    ...
}
```

Solution

If this is not an intended design, it is recommended to remove the redundant input parameter.

Status

Acknowledged

[N12] [Medium] Potential risks of using different participation rates

Category: Design Logic Audit

Content

In the BoringVaultAdapter contract, the previewDeposit function is used to calculate how many shares a user would receive for their deposited assets. It performs this calculation using the lens contract's previewDeposit function. However, it's important to note that the lens contract's previewDeposit function directly uses the accountant contract's exchangeRate for calculation without verifying whether the user's asset is the base token of the accountant contract. This could result in the rate used by the BoringVaultAdapter contract's previewDeposit function being different from the rate used by convertToShares.

Code location: nest/src/token/BoringVaultAdapter.sol#L346,L375

```
    function previewDeposit(
        uint256 assets
    ) public view virtual override(ComponentToken) returns (uint256) {
        BoringVaultAdapterStorage storage $ = _getBoringVaultAdapterStorage();

        return $.boringVault.lens.previewDeposit(
            IERC20(address($.asset)), assets, $.boringVault.vault,
$.boringVault.accountant
        );
    }

    function convertToShares(
        uint256 assets
    ) public view virtual override(ComponentToken) returns (uint256 shares) {
        BoringVaultAdapterStorage storage $ = _getBoringVaultAdapterStorage();

        try $.boringVault.vault.decimals() returns (uint8 shareDecimals) {
            shares = assets.mulDivDown(10 ** shareDecimals,
$.boringVault.accountant.getRateInQuote(ERC20(asset())));
        } catch {
            revert InvalidVault();
        }
    }
```

Solution

It is recommended to use the same logic as convertToShares for calculations to avoid this inconsistency.

Status

Fixed; Fixed in PR 115

[N13] [Low] Invalid balance acquisition

Category: Design Logic Audit

Content

In the BoringVaultAdapter contract, the balanceOf and assetsOf functions are used to query a user's shares and asset amounts in the vault contract. However, it's important to note that when BoringVaultAdapter deposits into the vault contract through the teller contract, the vault contract records BoringVaultAdapter as the depositor, not the actual user. This means that the vault contract's shares are held by the BoringVaultAdapter contract, while users hold shares minted by the BoringVaultAdapter contract. Consequently, when users query through the BoringVaultAdapter

contract's balanceOf and assetsOf functions, the returned amounts do not reflect their actual deposits in the BoringVaultAdapter contract.

Code location: nest/src/token/BoringVaultAdapter.sol#L428-L445

```
    function balanceOf(
        address account
    ) public view override(IERC20, ERC20Upgradeable) returns (uint256) {
        BoringVaultAdapterStorage storage $ = _getBoringVaultAdapterStorage();
        return $.boringVault.lens.balanceOf(account, $.boringVault.vault);
    }

    function assetsOf(
        address account
    ) public view virtual override(ComponentToken) returns (uint256) {
        BoringVaultAdapterStorage storage $ = _getBoringVaultAdapterStorage();
        return $.boringVault.lens.balanceOfInAssets(account, $.boringVault.vault,
$.boringVault.accountant);
    }
```

Solution

If this is not an intended design, it is recommended to modify these functions to return the user's actual shares and asset amounts deposited in the BoringVaultAdapter contract.

Status

Fixed; Fixed in PR 116

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002412040002	SlowMist Security Team	2024.12.03 - 2024.12.04	Medium Risk

Summary conclusion: The SlowMist security team uses a manual and SlowMist team's analysis tool to audit the project, during the audit work we found 2 critical risks, 4 medium risks, 2 low risks, 4 suggestions, and 1 information. All the findings were fixed or acknowledged. The code was not deployed to the mainnet. Since the project team has

not yet addressed the risk of excessive privileges for privileged roles, the current audit conclusion remains at medium risk.

6 Statement

SlowMist issues this report with reference to the facts that have occurred or existed before the issuance of this report, and only assumes corresponding responsibility based on these.

For the facts that occurred or existed after the issuance, SlowMist is not able to judge the security status of this project, and is not responsible for them. The security audit analysis and other contents of this report are based on the documents and materials provided to SlowMist by the information provider till the date of the insurance report (referred to as "provided information"). SlowMist assumes: The information provided is not missing, tampered with, deleted or concealed. If the information provided is missing, tampered with, deleted, concealed, or inconsistent with the actual situation, the SlowMist shall not be liable for any loss or adverse effect resulting therefrom. SlowMist only conducts the agreed security audit on the security situation of the project and issues this report. SlowMist is not responsible for the background and other conditions of the project.

Official Website

www.slowmist.com

E-mail

team@slowmist.com

Twitter

@SlowMist_Team

Github

https://github.com/slowmist