Smart Contract Security Audit Report

Table Of Contents

1 Executive Summary

On 2025.10.09, the SlowMist security team received the Mu Digital team's security audit application for Mu Protocol, 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

Mu Digital Protocol is a comprehensive DeFi protocol that primarily provides primary market minting and redemption services for stablecoins (AZND/MuBOND). It combines staking escrow and reward distribution mechanisms, enables the exchange between various stablecoins through price oracles, and facilitates the custody and withdrawal of funds through a treasury manager.

3.2 Vulnerability Information

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

NO	Title	Category	Level	Status
N1	Risk of excessive authority	Authority Control Vulnerability Audit	High	Acknowledged
N2	Reliance on a Centralized Oracle System Allows for Price Manipulation	Unsafe External Call Audit	High	Acknowledged
N3	Systemic Inflation Attack from Vault Logic Combined with Public Staking Entry Point	Design Logic Audit	Medium	Fixed
N4	First Price Update Bypasses Volatility Check	Design Logic Audit	Medium	Acknowledged
N5	Unbounded Redeem Requests Array Leads to Gas Exhaustion and Denial of Service	Denial of Service Vulnerability	Low	Fixed
N6	Lack of Input Validation on decimals Parameter Can Cause Transaction Reverts	Design Logic Audit	Low	Fixed
N7	Token Compatibility Reminder	Others	Suggestion	Acknowledged
N8	Allowance Management Logic May Cause Stake Transactions to Fail Unexpectedly	Others	Suggestion	Acknowledged
N9	Redemption Burns Tokens Before Verifying Sufficient Treasury Liquidity	Others	Suggestion	Acknowledged
N10	Preemptive Initialization	Race Conditions Vulnerability	Suggestion	Acknowledged
N11	Missing zero address check	Others	Suggestion	Acknowledged
N12	Front-Running Arbitrage Risk on Reward Distribution	Others	Information	Acknowledged

NO	Title	Category	Level	Status
N13	Swap-pop in claimRedeem invalidates pending request IDs causing user confusion	Others	Information	Acknowledged

4 Code Overview

4.1 Contracts Description

Audit Version:

https://github.com/Mu-Digital/mu-protocol

commit: c3bc589923aebf8f933d4432a1ed45619946bcf6

Fixed Version:

https://github.com/Mu-Digital/mu-protocol

commit: 261ccb457f7849787623c90ebbbe268514fefb33

Audit Scope:

```
./contracts
├── AccessManager.sol
├── commons
│   ├── ContractBaseUpgradeable.sol
│   ├── ContractUUPSUpgradeable.sol
│   └── Errors.sol
├── interfaces
│   ├── IAccessManager.sol
│   ├── IMuBONDPriceFeed.sol
│   ├── IMuSDPriceFeed.sol/(IAZNDPriceFeed.sol)
│   ├── IMuToken.sol
│   ├── IPriceFeed.sol
│   ├── IPrimaryMarket.sol
│   ├── IRewardDistributor.sol
│   ├── ISMuToken.sol/(ILoAZND.sol)
│   ├── IStakingEscrow.sol
│   └── ITreasuryManager.sol
├── oracles
│   ├── MuBONDPriceFeed.sol
```

```
│   └── MuSDPriceFeed.sol / (AZNDPriceFeed.sol)
├── PriceFeed.sol
├── PrimaryMarket.sol
├── proxies
│   └── ERC1967Proxy.sol
├── RewardDistributor.sol
├── StakingEscrow.sol
├── Timelock.sol
├── tokens
│   ├── MuBOND.sol
│   ├── MuSD.sol/(AZND.sol)
│   └── SMuSD.sol/(LoAZND.sol)
└── TreasuryManager.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:

TreasuryManager			
Function Name	Visibility	Mutability	Modifiers
initialize	External	Can Modify State	initializer
withdrawFund	External	Can Modify State	onlyWithdrawer
transferToCustodian	External	Can Modify State	onlyTransfer
_transferFund	Internal	Can Modify State	-

AccessManager			
Function Name	Visibility	Mutability	Modifiers
initialize	External	Can Modify State	initializer
getHashRole	Public	-	-
grantRoleForMultiAccounts	External	Can Modify State	onlyRole

AccessManager			
revokeRoleForMultiAccounts	External	Can Modify State	onlyRole

ContractUUPSUpgradeable			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
_authorizeUpgrade	Internal	Can Modify State	onlyRole

ContractBaseUpgradeable			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
_initContractBaseUpgradeable	Internal	Can Modify State	onlyInitializing
_authorizeUpgrade	Internal	Can Modify State	onlyUpgradableAdmin
setAccessManager	External	Can Modify State	onlyAdmin
pause	External	Can Modify State	onlyPauser
unpause	External	Can Modify State	onlyPauser

MuBONDPriceFeed			
Function Name	Visibility	Mutability	Modifiers
initialize	External	Can Modify State	initializer
setPriceFeed	External	Can Modify State	onlyAdmin
getPrice	External	-	-

MuSDPriceFeed			
Function Name	Visibility	Mutability	Modifiers

MuSDPriceFeed			
initialize	External	Can Modify State	initializer
setPriceFeed	External	Can Modify State	onlyAdmin
getPrice	External	-	-

MuBOND			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	ERC20
setAccessManager	External	Can Modify State	onlyAdmin
mint	External	Can Modify State	onlyMinter
burn	External	Can Modify State	onlyBurner

MuSD			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	ERC20
setAccessManager	External	Can Modify State	onlyAdmin
mint	External	Can Modify State	onlyMinter
burn	External	Can Modify State	onlyBurner

SMuSD			
Function Name	Visibility	Mutability	Modifiers
initialize	External	Can Modify State	initializer
deposit	Public	Can Modify State	onlyMinter
mint	Public	Can Modify State	onlyMinter
withdraw	Public	Can Modify State	onlyBurner

SMuSD			
redeem	Public	Can Modify State	onlyBurner
depositRewards	External	Can Modify State	onlyRewardDepositor
totalAssets	Public	-	-

PriceFeed			
Function Name	Visibility	Mutability	Modifiers
initialize	External	Can Modify State	initializer
updatePrice	External	Can Modify State	onlyOracle
getPrice	External	-	-
getFreshPrice	External	-	-
setMaxPriceChangePercent	External	Can Modify State	onlyAdmin
setStalenessThreshold	External	Can Modify State	onlyAdmin
setMinUpdateInterval	External	Can Modify State	onlyAdmin

PrimaryMarket			
Function Name	Visibility	Mutability	Modifiers
initialize	External	Can Modify State	initializer
setDepositedTokens	External	Can Modify State	onlyAdmin
setReceivedTokens	External	Can Modify State	onlyAdmin
addWhiteList	External	Can Modify State	onlyAdmin
removeWhiteList	External	Can Modify State	onlyAdmin
setTreasury	External	Can Modify State	onlyAdmin

PrimaryMarket			
setPriceFeed	External	Can Modify State	onlyAdmin
_calculateTokenAmount	Internal	Can Modify State	-
mint	External	Can Modify State	onlyWhiteList whenNotPaused nonReentrant
redeem	External	Can Modify State	onlyWhiteList whenNotPaused nonReentrant
retrieveERC20Tokens	External	Can Modify State	onlyAdmin

RewardDistributor			
Function Name	Visibility	Mutability	Modifiers
initialize	External	Can Modify State	initializer
schedule	External	Can Modify State	onlyAdmin
cancelSchedule	External	Can Modify State	onlyAdmin
rescheduleRemaining	External	Can Modify State	onlyAdmin
setVault	External	Can Modify State	onlyAdmin
sweepDust	External	Can Modify State	onlyAdmin
tick	External	Can Modify State	whenNotPaused nonReentrant
getSchedule	External	-	-
remaining	Public	-	-
pendingInstallments	Public	-	-

StakingEscrow			
Function Name	Visibility	Mutability	Modifiers
initialize	External	Can Modify State	initializer

StakingEscrow			
setVault	External	Can Modify State	onlyAdmin
setCoolDownPeriod	External	Can Modify State	onlyAdmin
addWhiteList	External	Can Modify State	onlyAdmin
removeWhiteList	External	Can Modify State	onlyAdmin
stake	External	Can Modify State	whenNotPaused nonReentrant
redeem	External	Can Modify State	whenNotPaused nonReentrant
claimRedeem	External	Can Modify State	whenNotPaused nonReentrant
getRedeemRequests	External	-	-

Timelock			
Function Name	Visibility	Mutability	Modifiers
initialize	Public	Can Modify State	initializer
supportsInterface	Public	-	-

4.3 Vulnerability Summary

[N1] [High] Risk of excessive authority

Category: Authority Control Vulnerability Audit

Content

These contracts of this protocol use AccessManager as the permission management contract. Through DEFAULT_ADMIN_ROLE, arbitrary roles can be configured for any contract, creating risks of excessive authority concentration.

1.PriceFeed Contract

The MANAGER_PRICE_FEED role can modify critical price parameters such as maxPriceChangePercent, stalenessThreshold, and minUpdateInterval. The ORACLE_ROLE can set arbitrary prices

and decimals with only basic validation (non-zero price).

Code Locations:

contracts/PriceFeed.sol#L85-L137, L168-L193

2.PrimaryMarket Contract

The MANAGER_PRIMARY_MARKET role can arbitrarily change which tokens are accepted for deposits/redemptions, complete control over user access without any checks or balances, redirect all future deposits to any address, change the priceFeed oracle contract, and extract any ERC20 tokens from the contract.

Code Locations:

contracts/PrimaryMarket.sol#L80-L159, L266-L272

3.SMuSD Contract

The TOKEN_VAULT_MINTER_ROLE can mint unlimited SMuSD tokens without economic restrictions. The TOKEN_VAULT_BURNER_ROLE can burn tokens from any user's account (with approval). The REWARD_DEPOSITOR_ROLE can inflate totalDepositedAssets without minting corresponding shares and enables potential ERC4626 inflation.

Code Locations:

contracts/tokens/SMuSD.sol#L87-L156

4.StakingEscrow Contract

The MANAGER_STAKING_ESCROW role controls all critical staking parameters, can change vault mappings for tokens, set cooldown periods to extreme values, and controls which addresses can bypass cooldown periods.

Code Locations:

contracts/StakingEscrow.sol#L66-L119

5.RewardDistributor Contract

The MANAGER_REWARD_DISTRIBUTOR role has complete control over reward distribution, can create reward schedules with arbitrary amounts and intervals, can cancel active schedules and redirect remaining funds to any address, can modify existing schedules, can redirect all future rewards to different vault, and can extract any tokens from the contract.

Code Locations:

contracts/RewardDistributor.sol#L64-L162

6.MuBONDPriceFeed and MuSDPriceFeed Contracts

The MANAGER_MUBOND_PRICE_FEED and MANAGER_MUSD_PRICE_FEED roles have unchecked control over critical price feed infrastructure.

Code Locations:

contracts/oracles/MuBONDPriceFeed.sol#L53-L57

contracts/oracles/MuSDPriceFeed.sol#L53-L57

7.MuBOND and MuSD Contracts

The TOKEN_MINTER_ROLE can mint unlimited MuBOND and MuSD tokens to any address without economic restrictions.

Code Locations:

contracts/tokens/MuBOND.sol#L81-L83

contracts/tokens/MuSD.sol#L81-L83

8.TreasuryManager Contract

The TREASURY_WITHDRAW_ROLE can withdraw any amount of any token to any address. The TREASURY_TRANSFER_ROLE can transfer unlimited funds to custodians

Code Locations:

contracts/TreasuryManager.sol#L63-L83

9.ContractBaseUpgradeable and ContractUUPSUpgradeable contract

Almost all contracts inherit ContractUUPSUpgradeable or ContractBaseUpgradeable contracts, where the admin role can modify the accessManager contract. Furthermore, the DEFAULT_ADMIN_ROLE role of accessManager can upgrade any contract.

Code location:

contracts/commons/ContractBaseUpgradeable.sol#L55, L85-L89

contracts/commons/ContractUUPSUpgradeable.sol#L25

Solution

In the short term, transferring owner ownership to multisig contracts is an effective solution to avoid single-point risk. But in the long run, it is a more reasonable solution to implement a privilege separation strategy and set up multiple privileged roles to manage each privileged function separately. And the authority involving user funds should be

managed by the community, and the EOA address can manage the authority involving emergency contract suspension. This ensures both a quick response to threats and the safety of user funds.

Status

Acknowledged; After communicating with the project team, they stated this is by design. The protocol intentionally uses a centralized authority structure through DEFAULT_ADMIN_ROLE and MANAGER roles for operational efficiency and flexibility during the initial phases. We have implemented robust safeguards through our Timelock contract, which holds the DEFAULT_ADMIN_ROLE and MANAGER_ROLE after initial setup. This ensures that all critical contract changes, role grants/revokes, and upgrades require multisig approval and a mandatory delay for community review.

[N2] [High] Reliance on a Centralized Oracle System Allows for Price Manipulation

Category: Unsafe External Call Audit

Content

The PrimaryMarket contract is critically dependent on the integrity of prices from the PriceFeed contract. The PriceFeed contract has key safety features: the maxPriceChangePercent, minUpdateInterval, and stalenessThreshold parameters, which are set by an Admin (MANAGER_PRICE_FEED) and are intended to prevent an Oracle (ORACLE_ROLE) from submitting a drastically different price. However, this entire safety model relies on the Admin and external Oracle. A malicious or compromised Admin can call setMaxPriceChangePercent to set the allowable change to 100%, effectively disabling this safeguard. Subsequently, a malicious or compromised Oracle can submit any arbitrary price to the PriceFeed. Since the PrimaryMarket contract trusts the PriceFeed, it will use this malicious price for its mint and redeem calculations, allowing an attacker to drain the treasury by minting tokens for nearly free or redeeming them for an exorbitant amount of collateral.

Code location:

contracts/PriceFeed.sol#L85-L151

contracts/PrimaryMarket.sol#L175-L176

```
    function updatePrice(address token, uint256 price, uint8 decimals) external
onlyOracle {
        ...
    }
```

```
    function getPrice(address token) external view returns (uint256 priceMantissa,
uint8 priceDecimals) {
        ...
    }

   function _calculateTokenAmount(
        ...t
    ) internal returns (uint256 amountOut) {
        (uint256 tokenInPriceMantissa, uint8 tokenInPriceDecimals) =
IPriceFeed(priceFeed).getPrice(tokenIn);
        (uint256 tokenOutPriceMantissa, uint8 tokenOutPriceDecimals) =
IPriceFeed(priceFeed).getPrice(tokenOut);
        ...
    }
```

Solution

It's recommended to use a manipulation-resistant oracle as the ORACLE_ROLE and secure the privileged roles.

Status

Acknowledged; After communicating with the project team, they stated that they set a default maxPriceChangePercent of 10% (1000 basis points) when deploying the PriceFeed contract, the deployment script to pass 1000 as default function, ensuring the contract launches with this value as intended. For the admin role, the plan is to implement Fireblocks MPC to secure the MANAGER_PRICE_FEED role, enhancing its security through multi-party computation. Regarding the use of a manipulation-resistant oracle, as the oracle implementation is not yet in place, this recommendation will be noted for future integration to ensure robust price feed security.

[N3] [Medium] Systemic Inflation Attack from Vault Logic Combined with Public Staking Entry Point

Category: Design Logic Audit

Content

Despite the SMuSD vault inheriting from a secure ERC4626 implementation in OpenZeppelin Contracts v5.4.0, the SMuSD and StakingEscrow contracts, when combined, create a system that is critically vulnerable to the ERC4626 First Deposit Inflation Attack through multiple vectors. This vulnerability arises from a fatal combination of incorrect function overrides in the vault and an exposed, unprotected staking entry point in the StakingEscrow contract.

The core issues are threefold:

1.Flawed Vault Logic (Root Cause): The SMuSD contract critically overrides the standard totalAssets() function to return a manually tracked totalDepositedAssets variable. This erroneous override directly bypasses and disables the

core security mechanism (i.e., the 'virtual share/asset' protection) inherited from the secure OpenZeppelin v5.4.0 base contract. The protection offered by the secure library is thus rendered completely ineffective, decoupling the vault's accounting from its actual asset balance.

2.External Attack Vector: The StakingEscrow.stake() function is external and lacks a minimum deposit amount. This allows any external attacker to call it with a trivial amount (e.g., 1 wei) to become the first depositor in an empty SMuSD vault. This action successfully establishes the prerequisite state (totalSupply = 1) for an inflation attack.

It is crucial to note that the traditional follow-up step of a "donation" (direct transfer to the vault) is ineffective for the attacker's personal profit under SMuSD's flawed design. The overridden totalAssets() ignores the donated funds, causing them to be permanently locked.

3. Internal Attack Vector: The flawed vault logic also creates a direct attack path for a malicious insider holding both MINTER and REWARD_DEPOSITOR roles. They can deposit 1 wei, then call depositRewards() to directly manipulate totalDepositedAssets, achieving the same share price inflation with even greater control.

Step 1 (Initial Deposit): The attacker uses their MINTER role to deposit a trivial amount (e.g., 1 wei), acquiring the vault's first share.

Step 2 (Value Inflation): The attacker then uses their REWARD_DEPOSITOR role to call depositRewards with a large sum of assets. Because this function directly increments the flawed totalDepositedAssets variable, it provides a "legitimate" and powerful way to instantly and massively increase the numerator of the share price calculation (totalAssets / totalSupply), making it the most efficient internal method for executing the inflation attack.

In addition, if there is a mistake in transferring MuSD tokens to the vault, the mistakenly transferred MuSD will be permanently locked in the contract.

Code location:

contracts/tokens/SMuSD.sol#L149-L164

```
function depositRewards(uint256 assets) external onlyRewardDepositor {
     if (assets == 0) revert Errors.InvalidAmount();
     address depositor = _msgSender();
     IERC20(asset()).safeTransferFrom(depositor, address(this), assets);
     totalDepositedAssets += assets;

     emit RewardDeposited(depositor, assets);
 }
```

```
    function totalAssets() public view override returns (uint256) {
        return totalDepositedAssets;
    }

   function stake(address token, uint256 amount) external whenNotPaused nonReentrant
returns (uint256) {
        ...
        uint256 amountOut = ISMuToken(tokenVault).deposit(actualAmount, sender);
        emit Staked(sender, token, tokenVault, actualAmount);
        return amountOut;
    }
```

Solution

It's recommended to remove the totalDepositedAssets state variable and the overridden totalAssets() function entirely. This will allow the contract to properly inherit and utilize the standard, secure implementation from the OpenZeppelin ERC4626Upgradeable v5.4.0 contract, re-enabling its built-in security mechanisms. And add a mandatory minimum deposit amount check to the stake function.

Status

Fixed

[N4] [Medium] First Price Update Bypasses Volatility Check

Category: Design Logic Audit

Content

In the PriceFeed contract, the updatePrice function contains a price volatility check that is wrapped inside an if (currentPrice > 0) condition. This means the check is completely skipped for the very first price update of any given token (when its currentPrice is 0). A malicious or compromised oracle can exploit this to set an initial price to an arbitrary and manipulative value (e.g., extremely high or low). Protocols that immediately begin using this price feed for the new token would be vulnerable to severe financial exploits, such as unfair liquidations, minting worthless assets, or draining funds through swaps.

Code location:

contracts/PriceFeed.sol#L85-L137

```
   function updatePrice(address token, uint256 price, uint8 decimals) external
onlyOracle {
        ...
```

```
        // If there's a previous price, check price change limit
        if (currentPrice > 0) {
            // Normalize both prices to the same decimal scale for comparison
            uint256 normalizedCurrentPrice;
            uint256 normalizedNewPrice;

            if (currentDecimals == decimals) {
                ...
            } else if (currentDecimals < decimals) {
                ...
            } else {
                ...
            }
            ...
            if (priceChange > maxAllowedChange) {
                revert Errors.PriceChangeExceedsLimit();
            }
        }
        // Update price data
        priceData.price = price;
        priceData.decimals = decimals;
        priceData.lastUpdate = block.timestamp;
        ...
    }
```

Solution

It's recommended to establish a more secure process for initializing prices or require the first price to be validated against a trusted, external on-chain source oracle.

Status

Acknowledged; After communicating with the project team, they stated that the MFC admin role, and manipulation-resistant oracle as the ORACLE_ROLE are believed could protect the price.

[N5] [Low] Unbounded Redeem Requests Array Leads to Gas Exhaustion and Denial of Service

Category: Denial of Service Vulnerability

Content

In the StakingEscrow contract, the redeem function adds a new RedeemRequest struct to the end of the redeemRequests array for every call. The contract does not enforce any limit on the number of redeem requests a user can create. A malicious actor could repeatedly call the redeem function with minimal amounts, causing the array for their account to grow indefinitely. This has several negative consequences:

1.Denial of Service (DoS): The getRedeemRequests view function, which is likely used by front-ends to display user information, will become prohibitively expensive to call as the array grows, effectively making it unusable.

2.Storage Bloat: Claimed requests are never removed, only marked with isClaimed = true, causing the contract's storage to grow permanently and unnecessarily.

Future Upgrade Risk: Any future contract function that might need to iterate over this array would be unexecutable due to gas limits.

Code location:

contracts/StakingEscrow.sol#L152-L211

```
   function redeem(address tokenVault, uint256 amount) external whenNotPaused
nonReentrant returns (uint256) {
        ...
        redeemRequests[spender][tokenVault].push(RedeemRequest(isClaimed, asset,
amount, amountOut, unlockTime));
        ...
    }

    function claimRedeem(address tokenVault, uint256 requestId) external
whenNotPaused nonReentrant returns (uint256) {
        ...
        RedeemRequest memory request = requests[requestId];
        ...
    }

    function getRedeemRequests(address account, address tokenVault) external view
returns (RedeemRequest[] memory) {
        return redeemRequests[account][tokenVault];
    }
```

Solution

It's recommended to prevent the redeem requests array from growing indefinitely by removing the claimed request from the array instead of only marking it or enforcing a limit on the number of pending redeem requests per user.

Status

Fixed

[N6] [Low] Lack of Input Validation on decimals Parameter Can Cause Transaction Reverts

Category: Design Logic Audit

Content

In the PriceFeed contract, the updatePrice function accepts a decimals parameter of type uint8 without validating if the value falls within a reasonable range. The contract's price normalization logic calculates a scaling factor by taking 10 to the power of the difference between the new and previous decimals values. If an operator provides a high initial decimals value (e.g., 100) and a subsequent update uses a low value (e.g., 18), the calculation 10 ** (100 - 18) will attempt to compute 10**82. This result exceeds the maximum value for a uint256 variable, causing a native arithmetic overflow which will cause the entire transaction to revert. This effectively creates a Denial of Service scenario where the price for a specific token can no longer be updated.

Code location:

contracts/PriceFeed.sol#L85-L137

```
    function updatePrice(address token, uint256 price, uint8 decimals) external
onlyOracle {
        ...
        if (currentPrice > 0) {
            ...
            if (currentDecimals == decimals) {
                ...
            } else if (currentDecimals < decimals) {
                // Scale up current price to match new decimals
                uint256 scaleFactor = 10 ** (decimals - currentDecimals);
                normalizedCurrentPrice = Math.mulDiv(currentPrice, scaleFactor, 1);
                normalizedNewPrice = price;
            } else {
                // Scale up new price to match current decimals
                uint256 scaleFactor = 10 ** (currentDecimals - decimals);
                normalizedCurrentPrice = currentPrice;
                normalizedNewPrice = Math.mulDiv(price, scaleFactor, 1);
            }
            ...
        }
        ...
    }
```

Solution

It's recommended to add a require statement at the beginning of the updatePrice function to constrain the decimals parameter to a reasonable range.

Status

Fixed

[N7] [Suggestion] Token Compatibility Reminder

Category: Others

Content

In the StakingEscrow contract, the stake function is designed with a two-step token transfer: first from the user to the StakingEscrow contract, and then from the StakingEscrow contract to the underlying vault. While the contract correctly uses a balance-before-and-after check to determine the actualAmount received in the first step, it fails to account for a potential second fee during the transfer to the vault. If the staked token charges a fee on every transfer, a fee will be deducted again when the vault pulls the funds from the StakingEscrow contract. This creates a mismatch where the vault's accounting is based on a larger asset amount than it actually received, causing the vault to become under-collateralized and leading to a loss of funds for the last users to withdraw.

Code location:

contracts/StakingEscrow.sol#L127-L144

```
    function stake(address token, uint256 amount) external whenNotPaused nonReentrant
returns (uint256) {
        ...
        // Use balance-before and balance-after pattern to handle fee-on-transfer
tokens
        uint256 balanceBefore = IERC20(token).balanceOf(address(this));
        IERC20(token).safeTransferFrom(sender, address(this), amount);
        uint256 balanceAfter = IERC20(token).balanceOf(address(this));
        uint256 actualAmount = balanceAfter - balanceBefore;
        uint256 allowanceAmount = IERC20(token).allowance(address(this), tokenVault);
        IERC20(token).safeIncreaseAllowance(tokenVault, actualAmount -
allowanceAmount);
        uint256 amountOut = ISMuToken(tokenVault).deposit(actualAmount, sender);
        ...
    }
```

Solution

It's recommended not to support such fee-on-transfer tokens or have the vault pull funds directly from the user after being approved by the user via the StakingEscrow contract.

Status

Acknowledged; The stake token mUSD used is not a fee-on-transfer token.

[N8] [Suggestion] Allowance Management Logic May Cause Stake Transactions to Fail Unexpectedly

Category: Others

Content

In the stake function, the logic to grant the vault an allowance to pull funds from the StakingEscrow contract is implemented by calculating the difference between the required amount and the current allowance, then increasing the allowance by that difference `safeIncreaseAllowance(tokenVault, actualAmount - allowanceAmount)`. This pattern is brittle because its success depends on the current state of the allowance. If due to a prior approval from a front-end or a previously failed transaction, the current allowanceAmount happens to be greater than the actualAmount to be deposited this time, the calculation actualAmount - allowanceAmount will fail due to an arithmetic underflow (which causes a revert in Solidity 0.8+). This will cause the user's stake transaction to unexpectedly fail, constituting a denial of service on user availability. The root cause of the failure can also be difficult for users and developers to diagnose.

Code location:

contracts/StakingEscrow.sol#L139-L40

```
    function stake(address token, uint256 amount) external whenNotPaused nonReentrant
returns (uint256) {
        ...
        uint256 allowanceAmount = IERC20(token).allowance(address(this), tokenVault);
        IERC20(token).safeIncreaseAllowance(tokenVault, actualAmount -
allowanceAmount);
        ...
    }
```

Solution

It is recommended to adopt the "approve-to-zero, then approve-to-amount" pattern.

Status

Acknowledged

[N9] [Suggestion] Redemption Burns Tokens Before Verifying Sufficient Treasury Liquidity

Category: Others

Content

The redeem function in the PrimaryMarket contract follows an order of operations where it first burns the user's tokenIn and then attempts to withdraw the corresponding tokenStableOut from the treasury contract. If the treasury contract has an insufficient balance of tokenStableOut to fulfill the redemption request. In this scenario, the final withdrawFund call will fail, causing the entire transaction to revert.

Code location:

contracts/PriceFeed.sol#L240-L258

```
    function redeem(
         address tokenIn,
         uint256 amountIn,
         address tokenStableOut,
         uint256 minAmountOut
     ) external onlyWhiteList whenNotPaused nonReentrant returns (uint256) {
         ...
         IMuToken(tokenIn).burn(receiver, amountIn);
         ITreasuryManager(treasury).withdrawFund(tokenStableOut, receiver,
amountStableOut);

         emit Redeemed(receiver, tokenIn, amountIn, tokenStableOut, amountStableOut);
         return amountStableOut;
     }
```

Solution

It’s recommended to add a balance check after calculating amountStableOut but before burning the user's tokens.

Status

Acknowledged

[N10] [Suggestion] Preemptive Initialization

Category: Race Conditions Vulnerability

Content

By calling the initialize function to initialize the contract, there is a potential issue that malicious attackers can preemptively call the initialize function to initialize.

Code location:

contracts/tokens/SMuSD.sol#L32-L43

contracts/AccessManager.sol#L16-L18

contracts/PriceFeed.sol#L43-L60

contracts/PrimaryMarket.sol#L52-l59

contracts/RewardDistributor.sol#L41-L47

contracts/StakingEscrow.sol#L44-L50

contracts/Timelock.sol#L20-L28

contracts/TreasuryManager.sol#L24-L28

```
    function initialize(address manager) external initializer {
         ...
     }
```

Solution

It is suggested that the initialization operation can be called in the same transaction immediately after the contract is created to avoid being maliciously called by the attacker.

Status

Acknowledged

[N11] [Suggestion] Missing zero address check

Category: Others

Content

In the AccessManager contract, when granting or revoking the DEFAULT_ADMIN_ROLE role, there is no check whether the address is 0.

Code location:

contracts/AccessManager#L34-L59

```
    function grantRoleForMultiAccounts(
         bytes32 role,
         address[] calldata accounts
     ) external onlyRole(getRoleAdmin(role)) {
         ...
     }
```

```
function revokeRoleForMultiAccounts(
    bytes32 role,
    address[] calldata accounts
) external onlyRole(getRoleAdmin(role)) {
    ...
}
```

Solution

It is recommended to add zero address check.

Status

Acknowledged

[N12] [Information] Front-Running Arbitrage Risk on Reward Distribution

Category: Others

Content

In the LoAZND reward distribution system, the timing and amount of reward distributions are publicly predictable via the RewardDistributor's getSchedule() function. Combined with the lack of time-weighted reward mechanisms, this allows sophisticated users (particularly whitelisted addresses) tcan monitor the mempool for RewardDistributor.tick() transactions, front-run with large deposits, and immediately withdraw after reward distribution to capture rewards disproportionate to their holding duration.

Mechanism:

When admin calls tick() to distribute rewards, the transaction is visible in the mempool before execution.

Attackers can:

Monitor mempool for tick() transactions.

Submit stake() transaction with higher gas to execute first.

Acquire shares at pre-reward price (lower cost per share).

Admin's tick() executes → depositRewards() increases all share values.

If whitelisted: immediately redeem() to extract profit.

Scenario demonstration:

Initial State (Before Attack):

LoAZND Vault:

totalAssets() = 200,000,000 AZND

totalSupply() = 200,000,000 LoAZND shares

(Price per share = 1.0 AZND)

Reward Signal: An Admin's tick() transaction is in the mempool, preparing to deposit 10,000,000 AZND as a reward.

Attacker: A user with 50,000,000 AZND of its own capital.

Attack Execution (Within a Single Block):

This user spots the tick() transaction and submits its own stake() transaction (Tx_Bot) with a higher gas fee.

Transaction 1 (Attacker): StakingEscrow.stake()

This user deposits 50,000,000 AZND at the pre-reward price of 1.0.

Successfully mints 50,000,000 LoAZND shares.

Vault state (interim): totalAssets = 250M, totalSupply = 250M.

Transaction 2 (Admin): RewardDistributor.tick()

The tick() function calls depositRewards(10,000,000).

The vault's totalAssets() increases by 10M, but totalSupply() does not change.

Vault state (final): totalAssets = 260,000,000, totalSupply = 250,000,000.

Arbitrage Result:

New Share Price: 260,000,000 (Assets) / 250,000,000 (Shares) = 1.04 AZND per share.

Attacker's Profit: The user's 50M shares are now worth 50,000,000 * 1.04 = 52,000,000 AZND.

The user has locked in 2,000,000 AZND in risk-free profit.

If this user is whitelisted, he can immediately withdraw without a cooldown period.

Impact:

Unfair reward distribution: Rewards intended for long-term stakers are diluted by short-term opportunistic entries.

Economic misalignment: System designed to incentivize long-term holding, but actually rewards short-term timing speculation.

Predictability exploitation: Public schedule information (nextRelease, installmentAmount) enables precise timing attacks.

Whitelist privilege risk: Whitelisted addresses can execute this strategy with minimal capital lockup risk.

Reduced long-term staker returns: Legitimate participants who provide stable liquidity receive diminished rewards.

Regular users partial protection:

Non-whitelisted users must wait coolDownPeriod before claiming redeemed assets

If coolDownPeriod spans multiple reward cycles, the arbitrage becomes less profitable (actor must hold through multiple distributions)

However, if the coolDownPeriod is short relative to reward frequency, arbitrage remains viable.

Code location:

contracts/RewardDistributor.sol#L174-L211

contracts/StakingEscrow.sol#131-198

```
    function tick(uint256 maxPayments) external onlyAdmin whenNotPaused nonReentrant
returns (uint256 released) {
        ...
        rewardToken.safeIncreaseAllowance(vault, released);
        ILoAZND(vault).depositRewards(released);
        ...
    }

    function stake(address token, uint256 amount) external whenNotPaused nonReentrant
returns (uint256) {
        ...
        uint256 amountOut = ILoAZND(tokenVault).deposit(actualAmount, sender);
        ...
    }

    function redeem(address tokenVault, uint256 amount) external whenNotPaused
nonReentrant returns (uint256) {
        ...
        bool isClaimed = isAddressInWhiteList[spender];
        uint256 unlockTime = isClaimed ? block.timestamp : block.timestamp +
coolDownPeriod;
        address receiver = isClaimed ? spender : address(this);
        uint256 balanceBefore = IERC20(asset).balanceOf(receiver);
        IERC20(tokenVault).safeTransferFrom(spender, address(this), amount);
        ILoAZND(tokenVault).redeem(amount, receiver, address(this));
        ...
        emit RedeemInitiated(spender, tokenVault, asset, amount, actualAmountReceived,
unlockTime);
        if (isClaimed) {
            ...
```

```
            );
        } else {
            redeemRequests[spender][tokenVault].push(RedeemRequest(asset, amount,
actualAmountReceived, unlockTime));
        }
        ...
    }
```

Solution

N/A

Status

Acknowledged

[N13] [Information] Swap-pop in claimRedeem invalidates pending request IDs causing user confusion

Category: Others

Content

The claimRedeem function uses swap-pop pattern to remove claimed requests from the array after the N5 fix. While this is gas-efficient, it causes the requestId (array index) of other pending requests to change unpredictably, leading to user experience issues and potential failed transactions. When a user claims a request that is not the last in the array, the function swaps the last element into the claimed position and pops the last element. This changes the array index (requestId) of the previously-last request, but users and front-ends may still reference the old index.

Example Scenario:

User has 4 pending requests: [req0, req1, req2, req3]

1.User calls claimRedeem(vault, 1) → Claims req1

- req3 moves from index 3 to index 1
- Array becomes: [req0, req3, req2]

2.User tries claimRedeem(vault, 3) → Reverts (index 3 no longer exists)

- User must query getRedeemRequests() again to find req3 is now at index 1

3.If user calls claimRedeem(vault, 1) again → Claims req3 (unexpected)

Impact:

User confusion: Saved requestIds become invalid after other claims

Failed transactions: Users attempting to claim by old index will revert

Front-end complexity: UI must refresh request list after every claim

No fund loss: Users can still claim all requests eventually after re-querying

Code location:

contracts/StakingEscrow.sol#L206-L226

```
    function claimRedeem(address tokenVault, uint256 requestId) external whenNotPaused
nonReentrant returns (uint256) {
        ...
        RedeemRequest[] storage requests = redeemRequests[receiver][tokenVault];
        if (requests.length == 0 || requestId > requests.length - 1) revert
Errors.InvalidRequestId();
        RedeemRequest memory request = requests[requestId];
        if (block.timestamp < request.unlockTime) revert Errors.CooldownTooShort();
        ...
        uint256 lastIndex = requests.length - 1;
        if (requestId != lastIndex) {
            // Move last element into the slot being removed
            requests[requestId] = requests[lastIndex];
        }
        requests.pop();
        ...
    }
```

Solution

Clearly document that requestId is an unstable array index, and instruct the front-end application to always re-fetch the entire getRedeemRequests array after any claimRedeem operation to display the correct, updated list to the user.

Status

Acknowledged; After communicating with the project team, they stated that their UI retrieve latest pending requests for user and it helps user to provide correct index in the request process.

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002510130001	SlowMist Security Team	2025.10.09 - 2025.10.13	High Risk

Summary conclusion: The SlowMist security team uses a manual and the SlowMist team's analysis tool to audit the project. During the audit work, we found 2 high risks, 2 medium risks, 2 low risks, 5 suggestions, and 2 information. All the findings were fixed or acknowledged. The code was not deployed to the mainnet.

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