Smart Contract Security Audit Report

Table Of Contents

1 Executive Summary

On 2025.11.03, the SlowMist security team received the BitxOnchain team's security audit application for FOF Fund, 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 a semi-decentralized fund management protocol, where all user assets are managed by privileged roles.

The audit primarily covers the whitelist, token, fund management, and fund operation modules. The whitelist module is mainly responsible for managing the protocol's whitelist users—only those on the whitelist are authorized to operate the fund within the protocol. The token module is used to mint fund certificates for users, with token creation handled by the token factory.

The fund management module is responsible for creating funds and managing their configurations. Whitelisted users can subscribe to or redeem funds through the fund operation module. Subscription funds are deposited with a third-party provider, and the provider's credentials are stored at the fund configuration's designated address.

The provider contract is not included in the audit scope and has not been open-sourced, so its internal risks remain unknown. Upon redemption, the funds will be returned from the third-party provider to the fund contract.

3.2 Vulnerability Information

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

NO	Title	Category	Level	Status
N1	Additional ownership transfer may cause initialization failure	Design Logic Audit	Medium	Fixed
N2	Redundant transfer return value check	Gas Optimization Audit	Suggestion	Fixed
N3	Some privileges of the FOFToken contract are locked	Authority Control Vulnerability Audit	Medium	Acknowledged
N4	There are unused variables in the Fund structure	Gas Optimization Audit	Suggestion	Fixed
N5	Redundant whitelist verification	Design Logic Audit	Suggestion	Fixed
N6	Missing `orderId` validation leading to potential request data overwrite	Design Logic Audit	High	Acknowledged
N7	Incorrect minting of liquidity-fee shares during purchase execution	Design Logic Audit	Information	Acknowledged
N8	Purchase operation relies on whitelist addresses to provide critical parameters	Authority Control Vulnerability Audit	Information	Acknowledged
N9	Precision loss due to division before multiplication in redemption amount calculation	Arithmetic Accuracy Deviation Vulnerability	Low	Acknowledged
N10	Inconsistency in fund calculation during	Design Logic Audit	Information	Acknowledged

NO	Title	Category	Level	Status
	redemption process			
N11	Redemption request does not follow the Checks-Effects-Interactions (CEI) principle	Design Logic Audit	Suggestion	Fixed
N12	Lack of cancellation mechanism	Design Logic Audit	Suggestion	Acknowledged
N13	Inconsistent NAV calculation basis between instant and delayed redemptions	Design Logic Audit	Information	Acknowledged
N14	Inconsistent fee deduction during redemption transfer	Design Logic Audit	Medium	Acknowledged
N15	Fee omission during cashValue update	Design Logic Audit	Medium	Acknowledged
N16	Dependency on non-open-source third-party contracts introduces unknown risks	Unsafe External Call Audit	Information	Acknowledged
N17	Potential risk due to missing reset step in approve operation	Denial of Service Vulnerability	Low	Acknowledged
N18	Excessive allowance to an unknown third-party contract causing risk exposure	Authority Control Vulnerability Audit	Low	Acknowledged
N19	Risk of excessive privilege	Authority Control Vulnerability Audit	Medium	Acknowledged
N20	Centralized ownership privileges and lack of role separation	Authority Control Vulnerability Audit	Suggestion	Acknowledged
N21	Missing initialization state check during fund creation	Design Logic Audit	Suggestion	Acknowledged

4 Code Overview

4.1 Contracts Description

Audit Version:

https://github.com/BitxOnchain/fof_fund_evm

commit: 3c935c908e702bca9836e52545e51dae97475baf

Fixed Version:

https://github.com/BitxOnchain/fof_fund_evm

commit: 3988013b98de73d89747cf979c52fa0e0cd6b201

Audit Scope:

```
./contract/contracts
├── core
│   ├── FundState.sol
│   ├── IFOFStorage.sol
│   ├── IFOFToken.sol
│   ├── IFOFTokenFactory.sol
│   ├── ISupplier.sol
│   └── WhitelistBaseUpgradeable.sol
├── logic
│   ├── FundManagementLogic.sol
│   └── FundOperationLogic.sol
├── storage
│   └── FOFStorage.sol
├── tokens
│   ├── FOFToken.sol
│   └── FOFTokenFactory.sol
└── utils
    └── Utils.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:

WhitelistBaseUpgradeable			
Function Name	Visibility	Mutability	Modifiers
__WhitelistBase_init	Internal	Can Modify State	onlyInitializing
addToWhitelist	External	Can Modify State	onlyOwner
removeFromWhitelist	External	Can Modify State	onlyOwner
batchAddToWhitelist	External	Can Modify State	onlyOwner
batchRemoveFromWhitelist	External	Can Modify State	onlyOwner

FOFStorage			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	Ownable
authorizeLogicContract	External	Can Modify State	onlyOwner
createFund	External	Can Modify State	onlyAuthorized
updateFund	External	Can Modify State	onlyAuthorized
isFundExists	Public	-	-
createPurchaseRequest	External	Can Modify State	onlyAuthorized
updatePurchaseRequest	External	Can Modify State	onlyAuthorized
createRedemptionRequest	External	Can Modify State	onlyAuthorized
updateRedemptionRequest	External	Can Modify State	onlyAuthorized
getPurchaseRequest	External	-	-
getRedemptionRequest	External	-	-
getFund	External	-	-

FOFToken			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
initialize	External	Can Modify State	initializer
decimals	Public	-	-
mint	External	Can Modify State	onlyMinter whenNotPaused nonReentrant notBlacklisted
burnFrom	Public	Can Modify State	onlyMinter whenNotPaused nonReentrant
transfer	Public	Can Modify State	whenNotPaused nonReentrant notBlacklisted notBlacklisted respectsCooldown onlyWhitelisted
transferFrom	Public	Can Modify State	whenNotPaused nonReentrant notBlacklisted notBlacklisted respectsCooldown onlyWhitelisted
addMinter	External	Can Modify State	onlyOwner
removeMinter	External	Can Modify State	onlyOwner
updateBlacklist	External	Can Modify State	onlyOwner
updateWhitelist	External	Can Modify State	onlyOwner
updateMaxSupply	External	Can Modify State	onlyOwner
updateTransferCooldown	External	Can Modify State	onlyOwner
pause	External	Can Modify State	onlyOwner
unpause	External	Can Modify State	onlyOwner
emergencyRecoverToken	External	Can Modify State	onlyOwner
isMinter	External	-	-

FOFToken			
isBlacklisted	External	-	-
getRemainingCooldown	External	-	-
getTokenInfo	External	-	-

FOFTokenFactory			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	Ownable
createFOFToken	External	Can Modify State	-
allTokensLength	External	-	-
setImplementation	External	Can Modify State	onlyOwner

FundManagementLogic			
Function Name	Visibility	Mutability	Modifiers
modifyFundAuthority	External	Can Modify State	onlyOwner
initialize	External	Can Modify State	initializer
setFofTokenFactory	External	Can Modify State	onlyOwner
setStorageAddress	External	Can Modify State	onlyOwner
setOperationLogic	External	Can Modify State	onlyOwner
initializeFund	External	Can Modify State	onlyWhitelisted whenNotPaused nonReentrant
updateNav	External	Can Modify State	onlyFundAuthority whenNotPaused nonReentrant
updateFeeConfig	External	Can Modify State	onlyFundAuthority

FundManagementLogic			
updateNavConfig	External	Can Modify State	onlyFundAuthority
pauseFund	External	Can Modify State	onlyFundAuthority
resumeFund	External	Can Modify State	onlyFundAuthority
forceUpdateNav	External	Can Modify State	onlyFundAuthority
forceMintFof	External	Can Modify State	onlyOwner
pause	External	Can Modify State	onlyOwner
unpause	External	Can Modify State	onlyOwner

FundOperationLogic			
Function Name	Visibility	Mutability	Modifiers
initialize	External	Can Modify State	initializer
setStorageAddress	External	Can Modify State	onlyOwner
setSupplierAddress	External	Can Modify State	onlyOwner
submitPurchaseRequest	External	Payable	fundExists onlyWhitelisted whenNotPaused nonReentrant
processPurchaseRequest	External	Can Modify State	onlyWhitelisted whenNotPaused nonReentrant
submitRedemptionRequest	External	Can Modify State	onlyWhitelisted fundExists whenNotPaused nonReentrant
processRedemptionRequest	External	Can Modify State	onlyWhitelisted whenNotPaused nonReentrant
setMinimumPurchaseAmount	External	Can Modify State	onlyOwner
setMinimumRedemptionShares	External	Can Modify State	onlyOwner

FundOperationLogic			
pause	External	Can Modify State	onlyOwner
unpause	External	Can Modify State	onlyOwner
getUserTokenBalance	External	-	-
_PurchaseToSupplier	Internal	Can Modify State	-
_RedemptionToSupplier	Internal	Can Modify State	-
_calculatePurchaseFee	Internal	-	-
_calculateSupplierPurchaseFee	Internal	-	-
_calculateRedemptionFee	Internal	-	-
_calculateSupplierRedemptionFee	Internal	-	-
_calculateLiquidityFee	Internal	-	-
_executePurchaseTransfers	Internal	Can Modify State	-

4.3 Vulnerability Summary

[N1] [Medium] Additional ownership transfer may cause initialization failure

Category: Design Logic Audit

Content

In the WhitelistBaseUpgradeable contract's `__WhitelistBase_init` function, which is responsible for initializing contract ownership and the whitelist, ownership is first set to `initialOwner` via `__Ownable_init(initialOwner)`, and then `transferOwnership(initialOwner)` is called again. Since the `transferOwnership` function has the `onlyOwner` modifier, it checks whether `msg.sender` is the current owner. At this point, the owner has already been set to `initialOwner`, so if the caller (`msg.sender`) is not `initialOwner`, the transaction will revert.

Code location: contract/contracts/core/WhitelistBaseUpgradeable.sol#L20

```
function __WhitelistBase_init(address initialOwner) internal onlyInitializing {
     __Ownable_init(initialOwner);
     transferOwnership(initialOwner);
     whitelist[initialOwner] = true;
     emit WhitelistUpdated(initialOwner, true);
 }
```

Solution

Remove the redundant `transferOwnership` operation from the `__WhitelistBase_init` function. The call to `__Ownable_init` alone is sufficient to complete ownership initialization.

Status

Fixed

[N2] [Suggestion] Redundant transfer return value check

Category: Gas Optimization Audit

Content

In the FOFToken contract's `transfer` and `transferFrom` functions, both override the standard ERC20 transfer and transfer-from logic to add extra controls (such as pause, blacklist, and cooldown features). After invoking the parent contract's `super.transfer` and `super.transferFrom`, the code performs an `if (success)` check on the returned `bool success` variable. This check is redundant because OpenZeppelin's ERC20 implementation reverts the transaction upon failure rather than returning `false`. Consequently, the code within the `if` block will only execute if the transfer succeeds, making the conditional check unnecessary and increasing gas consumption for each transfer.

Code location: contract/contracts/tokens/FOFToken.sol#L154,L177

```
function transfer(address to, uint256 amount)
     public
     override
     whenNotPaused
     nonReentrant
     notBlacklisted(msg.sender)
     notBlacklisted(to)
     respectsCooldown(msg.sender)
```

```
        onlyWhitelisted(msg.sender)
        returns (bool)
    {
        address from = msg.sender;


        bool success = super.transfer(to, amount);

        if (success) {
            lastTransfer[from] = block.timestamp;
        }
        return success;
    }

    function transferFrom(address from, address to, uint256 amount)
        public
        override
        whenNotPaused
        nonReentrant
        notBlacklisted(from)
        notBlacklisted(to)
        respectsCooldown(from)
        onlyWhitelisted(msg.sender)
        returns (bool)
    {

        bool success = super.transferFrom(from, to, amount);

        if (success) {
            lastTransfer[from] = block.timestamp;
        }

        return success;
    }
```

Solution

It is recommended to remove the redundant `success` condition check and the associated `if` statement.

Status

Fixed

[N3] [Medium] Some privileges of the FOFToken contract are locked

Category: Authority Control Vulnerability Audit

Content

In the FundManagementLogic contract's `initializeFund` function, when a new `FOFToken` instance is created through `fofTokenFactory.createFOFToken`, the `FundManagementLogic` contract itself (`address(this)`) is set as the owner of the newly created `FOFToken`. However, the `FundManagementLogic` contract does not expose any external or public functions to invoke the management actions in `FOFToken` that are restricted to the `onlyOwner` role (such as `pause`, `unpause`, `updateBlacklist`, `removeMinter`, and `emergencyRecoverToken` ...). This effectively locks the critical administrative privileges of the `FOFToken` within the contract, preventing administrators from performing necessary maintenance or emergency operations and thereby introducing potential governance risks.

Code location: contract/contracts/logic/FundManagementLogic.sol#L107

```
function initializeFund(
    ...
) external onlyWhitelisted whenNotPaused nonReentrant {
    ...
    address fofToken = fofTokenFactory.createFOFToken(
        string(abi.encodePacked("FOF ", fundName)),
        fundSymbol,
        address(this), // Set this contract as the owner
        10**32 // A very large number for max supply
    );
    ...
}
```

Solution

It is recommended to add functions within the `FundManagementLogic` contract, protected by appropriate access controls, to expose the `onlyOwner` management capabilities of the `FOFToken`. This would allow the contract owner to indirectly execute these administrative operations.

Status

Acknowledged; After communicating with the project team, the team stated that this was the intended design.

[N4] [Suggestion] There are unused variables in the Fund structure

Category: Gas Optimization Audit

Content

In the FundState contract, the `Fund` struct defines and stores the core state data for each fund. However, several

variables within the struct— `nextAllowedUpdateTimestamp`, `totalAssetValue`, and `last14StTokenNavList` —are neither read nor written in any part of the protocol's logic.

Code location: contract/contracts/core/FundState.sol#L71,L75,L81

```
struct Fund {
     ...
     uint256 nextAllowedUpdateTimestamp;
     ...
     uint256 totalAssetValue;
     ...
     NavValue[14] last14StTokenNavList;
     ...
 }
```

Solution

If these variables are not reserved for future use, it is recommended to remove `nextAllowedUpdateTimestamp`, `totalAssetValue`, and `last14StTokenNavList` from the `Fund` struct to optimize storage efficiency and improve code clarity.

Status

Fixed

[N5] [Suggestion] Redundant whitelist verification

Category: Design Logic Audit

Content

In the FundOperationLogic contract's `submitPurchaseRequest` function, which handles user fund subscription requests, the function already includes the `onlyWhitelisted` modifier that checks whether `msg.sender` is in the whitelist. However, within the function body, there is an additional `require` statement that again validates `whitelist[msg.sender]`. This extra check is redundant, as it performs the exact same verification as the `onlyWhitelisted` modifier.

Code location: contract/contracts/logic/FundOperationLogic.sol#L109,L122

```
function submitPurchaseRequest(
     string memory fundName,
     address user,
```

```
        uint256 amount,
        string memory orderId
    )
        external
        payable
        fundExists(fundName)
        onlyWhitelisted
        whenNotPaused
        nonReentrant
    {
        ...
        require(whitelist[msg.sender], "User not whitelisted");
        ...
    }
```

Solution

It is recommended to remove the redundant whitelist check inside the `submitPurchaseRequest` function body.

Status

Fixed

[N6] [High] Missing `orderId` validation leading to potential request data overwrite

Category: Design Logic Audit

Content

In the FundOperationLogic contract's `submitPurchaseRequest` and `submitRedemptionRequest` functions, when users submit purchase or redemption requests, the system uses the user-provided `orderId` as a unique identifier to create and store the request record. However, the current implementation does not verify whether the `orderId` has already been used. This allows a caller to submit a new purchase request with an existing `orderId`, thereby overwriting the previous record and disrupting normal fund operations.

The same applies to the submitRedemptionRequest function.

Code location: contract/contracts/logic/FundOperationLogic.sol#L197,L429

```
    function submitPurchaseRequest(
        ...
    )
        external
        payable
        fundExists(fundName)
```

```
        onlyWhitelisted
        whenNotPaused
        nonReentrant
    {
        ...
        fofStorage.createPurchaseRequest(orderId, request);

        ...
    }

    function submitRedemptionRequest(
        ...
    ) external onlyWhitelisted fundExists(fundName) whenNotPaused nonReentrant {
        ...
        fofStorage.createRedemptionRequest(orderId, request);

        ...
    }
```

Solution

It is recommended to add a uniqueness check for `orderId` within the `fofStorage.createPurchaseRequest` function, or alternatively validate the uniqueness of `orderId` before creating new requests.

Status

Acknowledged; After communicating with the project team, they stated that this is the intended design, and the protocol's server side can ensure that there will be no duplicate submissions of the same orderId.

[N7] [Information] Incorrect minting of liquidity-fee shares during purchase execution

Category: Design Logic Audit

Content

In the FundOperationLogic contract's `processPurchaseRequest` function, the base amount `cashAmount` used to calculate the number of FOF tokens to mint (`tokensToMint`) is derived as `request.cashAmount - request.onchainDeductedFee`. Here, `request.cashAmount` represents the user's total deposit, so `cashAmount` corresponds to the net amount after on-chain fees are deducted. However, during the `submitPurchaseRequest` process, a portion of the funds is allocated as `liquidityFee`, which is *not* included in `onchainDeductedFee`. This fee should belong to the protocol. The current logic incorrectly includes the `liquidityFee` in `cashAmount`, thereby minting FOF tokens for the user based on this amount. As a result,

liquidity fees intended as protocol revenue are effectively returned to users as additional fund shares, reducing the protocol's economic benefit.

Code location: contract/contracts/logic/FundOperationLogic.sol#L239-L246

```
function processPurchaseRequest(
    string memory orderId,
    uint256 realizedFundQuantity,
    uint256 mintNav
) external onlyWhitelisted whenNotPaused nonReentrant {
    ...
    uint256 cashAmount = request.cashAmount - request.onchainDeductedFee;
    // 计算要铸造的代币数量（对应 Solana 的 calculate_cash_to_tokens）
    uint256 tokensToMint = Utils.calculateTokenAmountByNav(
        cashAmount,
        mintNav,
        IERC20Metadata(fund.fofToken).decimals(),
        IERC20Metadata(fund.cashToken).decimals()
    );
    ...
    require(
        fund.inTransitIncomingCash >= request.liquidityFee,
        "Invalid in-transit cash"
    );
    fund.inTransitIncomingCash -= request.liquidityFee;

    ...
    fund.assetValue.cashValue += request.liquidityFee;
    ...
    FOFToken(fund.fofToken).mint(request.user, tokensToMint);

    ...
}
```

Solution

When calculating `tokensToMint`, the `liquidityFee` should be excluded from the base amount.

Status

Acknowledged; After communicating with the project team, they confirmed that this is the intended design.

[N8] [Information] Purchase operation relies on whitelist addresses to provide critical parameters

Category: Authority Control Vulnerability Audit

Content

In the FundOperationLogic contract's `processPurchaseRequest` function, which handles submitted purchase requests and mints fund shares (FOF Tokens) for users, the execution depends on two critical parameters provided by whitelist addresses — `realizedFundQuantity` and `mintNav`. This design introduces both a single-point-of-failure and an excessive-privilege risk.

The same applies to the processRedemptionRequest function.

Code location: contract/contracts/logic/FundOperationLogic.sol#L208-L211

```
function processPurchaseRequest(
    string memory orderId,
    uint256 realizedFundQuantity,
    uint256 mintNav
) external onlyWhitelisted whenNotPaused nonReentrant {
…
}
```

Solution

In the long term, the introduction of an oracle system to obtain net asset values and token quantities from trusted providers would effectively eliminate the excessive-privilege risk of whitelist users. In the short term, requiring whitelist users to operate through a multisignature wallet can mitigate single-point risks. Additionally, implementing real-time monitoring to verify whether whitelist users' actions align with expectations—and enabling emergency protocol suspension in case of anomalies—would further enhance security.

Status

Acknowledged; After communicating with the project team, they confirmed that this is the intended design.

[N9] [Low] Precision loss due to division before multiplication in redemption amount calculation

Category: Arithmetic Accuracy Deviation Vulnerability

Content

In the redemption handling process of the FundOperationLogic contract, when computing the final redemption amount `redeemFundAmount`, the code uses the expression `(tmpRedeemFundAmount / tempFundAssetPrecision) * tempFundAssetPrecision`. Because Solidity's integer division truncates

decimals (rounds down), this operation leads to precision loss.

Such precision loss is unfavorable to users initiating redemptions — they will receive slightly less than the amount they are actually entitled to based on their share, with the remaining fractional assets left in the contract.

Code location: contract/contracts/logic/FundOperationLogic.sol#L345-L348

```
function submitRedemptionRequest(
    string memory fundName,
    uint256 shares,
    string memory orderId,
    address user
) external onlyWhitelisted fundExists(fundName) whenNotPaused nonReentrant {
    ...
    uint256 tempFundAssetPrecision = fundAssetPrecision / 1000;
    //向supplier赎回redeemFundAmount的asset份额
    uint256 redeemFundAmount = (tmpRedeemFundAmount /
        tempFundAssetPrecision) * tempFundAssetPrecision;
    ...
}
```

Solution

Unless this is an intentional design choice, it is recommended to reassess the necessity of rounding down. Instead, use the unrounded `tmpRedeemFundAmount` directly in subsequent calculations and transfers to ensure users receive their full redemption value. If the rounding behavior is required for business reasons, the resulting “dust” difference should be explicitly handled and documented.

Status

Acknowledged; After communicating with the project team, they confirmed that this is the intended design.

[N10] [Information] Inconsistency in fund calculation during redemption process

Category: Design Logic Audit

Content

In the FundOperationLogic contract’s redemption handling function, when processing user redemption requests, the contract withdraws `redeemFundAmount` from the fund’s asset wallet (`fund.assetWallet`). This `redeemFundAmount` represents the net value obtained by deducting a 5% fee from the user’s total redeemable amount (`realizableCash`). However, in the subsequent transfer logic, the contract still transfers the original total

(`realizableCash`)—before fees—to the user.

This creates a funds mismatch: the contract only retrieves the net amount (95%) from the provider but pays out the full amount (100%) to the user.

Code location: contract/contracts/logic/FundOperationLogic.sol#L352-L381

```
function submitRedemptionRequest(
    string memory fundName,
    uint256 shares,
    string memory orderId,
    address user
) external onlyWhitelisted fundExists(fundName) whenNotPaused nonReentrant {
    ...
    uint256 redeemCash = (realizableCash * 95) / 100;
    // Calculate redemption fund amount based on cash amount and NAV
    uint256 fundAssetPrecision = 10 **
        IERC20Metadata(fund.fofToken).decimals();
    uint256 tmpRedeemFundAmount = Utils.calculateTokenAmountByNav(
        redeemCash,
        fund.assetNetAssetValue,
        IERC20Metadata(fund.assetToken).decimals(),
        IERC20Metadata(fund.cashToken).decimals()
    );

    uint256 tempFundAssetPrecision = fundAssetPrecision / 1000;
    //向supplier赎回redeemFundAmount的asset份额
    uint256 redeemFundAmount = (tmpRedeemFundAmount /
        tempFundAssetPrecision) * tempFundAssetPrecision;
    console.log("redeemFundAmount:", redeemFundAmount);

    //先把assetToken从assetWallet转到合约地址
    IERC20(fund.assetToken).safeTransferFrom(
        fund.assetWallet,
        address(this),
        redeemFundAmount
    );
    ...
    if (judgeIfRealize) {
        ...
        IERC20(fund.cashToken).safeTransferFrom(
            fund.assetWallet,
            user,
            realizableCash
        );

        ...
```

```
        } else {
            ...
        }
        ...
    }
```

Solution

Unless this behavior is intentional, it is essential to unify the accounting logic in the redemption process. The final amount paid to users should be the fee-deducted net value (`redeemCash`), not the original total (`realizableCash`).

Status

Acknowledged; After communicating with the project team, they confirmed that this is the intended design.

[N11] [Suggestion] Redemption request does not follow the Checks-Effects-Interactions (CEI) principle

Category: Design Logic Audit

Content

In the FundOperationLogic contract's redemption handling function, the order of operations does not comply with the Checks-Effects-Interactions principle. The function calls `safeTransferFrom` to transfer funds to the user and the fee wallet **before** updating internal state variables (such as invoking `burnFrom` to burn user shares and updating the fund's total supply `totalSupply`).

The same applies to the processRedemptionRequest function.

Code location: contract/contracts/logic/FundOperationLogic.sol#L370-L392

```
    function submitRedemptionRequest(
        string memory fundName,
        uint256 shares,
        string memory orderId,
        address user
    ) external onlyWhitelisted fundExists(fundName) whenNotPaused nonReentrant {
        ...
        if (judgeIfRealize) {
            //先收费
            uint256 fundFee = onchainDeductedFee - supplierDeductedFee;
            if (fundFee > 0) {
                //收费给fund.ocFeeWallet
```

```
                IERC20(fund.cashToken).safeTransferFrom(
                    fund.assetWallet,
                    fund.ocFeeWallet,
                    fundFee
                );
            }
            //向用户转账赎回的现金
            IERC20(fund.cashToken).safeTransferFrom(
                fund.assetWallet,
                user,
                realizableCash
            );

            //burn掉用户的fofToken
            FOFToken(fund.fofToken).burnFrom(user, shares);
            if (fund.totalSupply < shares) {
                revert PurchaseAmountTooLow(shares, fund.totalSupply);
            }
            if (fund.assetValue.cashValue < realizableCash + fundFee) {
                revert PurchaseAmountTooLow(realizableCash + fundFee,
fund.assetValue.cashValue);
            }
            fund.totalSupply -= shares;
            fund.assetValue.cashValue -= realizableCash + fundFee;
        } else {
            ...
        }
        ...
    }
```

Solution

Although this issue does not directly lead to reentrancy risk, it is still recommended to adhere to the CEI pattern — perform all internal state updates first and execute external fund transfers last.

Status

Fixed

[N12] [Suggestion] Lack of cancellation mechanism

Category: Design Logic Audit

Content

In the `FundOperationLogic` contract's `submitPurchaseRequest` and `submitRedemptionRequest` functions,

the constructed `FundState` includes an `isCanceled` field. However, the current process does not provide any interface or state transition to set this field.

Code location: contract/contracts/logic/FundOperationLogic.sol#L189,L416

```
function submitPurchaseRequest(
    string memory fundName,
    address user,
    uint256 amount,
    string memory orderId
)
    external
    payable
    fundExists(fundName)
    onlyWhitelisted
    whenNotPaused
    nonReentrant
{
    ...
    FundState.PurchaseRequest memory request = FundState.PurchaseRequest({
        ...
        isCanceled: false,
        ...
    });
    ...
}

function submitRedemptionRequest(
    string memory fundName,
    uint256 shares,
    string memory orderId,
    address user
) external onlyWhitelisted fundExists(fundName) whenNotPaused nonReentrant {
    ...
    FundState.RedemptionRequest memory request = FundState
        .RedemptionRequest({
            ...
            isCanceled: false,
            ...
        });
    ...
}
```

Solution

Unless this is an intentional design choice, it is recommended to implement cancellation logic to allow requests to be

withdrawn.

Status

Acknowledged; After communicating with the project team, they confirmed that this is the intended design.

[N13] [Information] Inconsistent NAV calculation basis between instant and delayed redemptions

Category: Design Logic Audit

Content

In the `FundOperationLogic` contract's redemption process, when a user triggers an instant redemption via `submitRedemptionRequest`, the redemption `cashAmount` is calculated based on the current `fund.fofNetAssetValue`. However, in the case of a delayed redemption, the final cash amount received by the user in `processRedemptionRequest` is determined using a `redeemNav` value provided by a whitelisted address. These two NAV values may differ, causing users redeeming the same number of shares at different times or under different liquidity conditions to receive unequal cash values — a situation that could potentially be exploited for arbitrage.

Code location: contract/contracts/logic/FundOperationLogic.sol#L309,L467

```
function processRedemptionRequest(
     string memory orderId,
     uint256 realizedCashoutAmount,
     uint256 redeemNav
 ) external onlyWhitelisted whenNotPaused nonReentrant {
     ...
     uint256 toUserCashAmount = Utils.calculateCashAmountByNav(
         request.tokenAmount,
         redeemNav,
         IERC20Metadata(fund.fofToken).decimals(),
         IERC20Metadata(fund.cashToken).decimals()
     );
     ...
 }
```

Solution

Unify the NAV calculation and application logic. For all redemption requests, the final settlement amount — whether processed instantly in `submitRedemptionRequest` or later in `processRedemptionRequest` — should be consistently based on a single, standardized NAV source.

Status

Acknowledged; After communicating with the project team, they confirmed that this is the intended design.

[N14] [Medium] Inconsistent fee deduction during redemption transfer

Category: Design Logic Audit

Content

In the FundOperationLogic.sol contract's `processRedemptionRequest` function, which handles redemption requests, the non-instant redemption path deducts fees calculated as `onchainDeductedFee - supplierDeductedFee` before transferring funds to the user. However, in the `submitRedemptionRequest` function's instant redemption path, the deduction is only `onchainDeductedFee`. This inconsistency causes the two redemption modes to apply different fee calculations and transfer amounts, potentially resulting in misallocated funds or over-/under-charged fees.

Code location: contract/contracts/logic/FundOperationLogic.sol#L332

```
function submitRedemptionRequest(
    string memory fundName,
    uint256 shares,
    string memory orderId,
    address user
) external onlyWhitelisted fundExists(fundName) whenNotPaused nonReentrant {
    ...
    uint256 realizableCash = cashAmount - onchainDeductedFee;
    ...
    if (judgeIfRealize) {
        ...
        IERC20(fund.cashToken).safeTransferFrom(
            fund.assetWallet,
            user,
            realizableCash
        );

        ...
    } else {
        ...
    }
    ...
}

function processRedemptionRequest(
```

```
        string memory orderId,
        uint256 realizedCashoutAmount,
        uint256 redeemNav
    ) external onlyWhitelisted whenNotPaused nonReentrant {
        ...
        uint256 fee = request.onchainDeductedFee - request.supplierDeductedFee;
        ...
        if (!request.ifRealize) {
            ...
            IERC20(fund.cashToken).safeTransferFrom(
                fund.assetWallet,
                request.user,
                toUserCashAmount - fee
            );
            ...
        }
        ...
    }
```

Solution

Unify the fee-deduction logic for both redemption types. For example, update the fee calculation in `processRedemptionRequest` to use `onchainDeductedFee` for consistency, ensuring uniform transfer amounts across both redemption flows.

Status

Acknowledged; After communicating with the project team, they confirmed that this is the intended design.

[N15] [Medium] Fee omission during cashValue update

Category: Design Logic Audit

Content

In the FundOperationLogic.sol contract's `processRedemptionRequest` function, which handles redemption requests, the update to `fund.assetValue.cashValue` subtracts only `toUserCashAmount` (the amount already deducted by fees) but does not add back the `fundFee` portion. In contrast, within the `submitRedemptionRequest` function, the `fund.assetValue.cashValue` update includes adding `fundFee`.

Code location: contract/contracts/logic/FundOperationLogic.sol#L509

```
    function processRedemptionRequest(
        string memory orderId,
        uint256 realizedCashoutAmount,
```

```
        uint256 redeemNav
    ) external onlyWhitelisted whenNotPaused nonReentrant {
        ...
            fund.totalSupply -= request.tokenAmount;
            fund.assetValue.cashValue -= toUserCashAmount;
        }

        ...
    }
```

Solution

Standardize the handling of `fundFee` when updating `fund.assetValue.cashValue`. Either add `fundFee` in both cases or omit it consistently to ensure accounting accuracy and uniform state updates.

Status

Acknowledged; After communicating with the project team, they confirmed that this is the intended design.

[N16] [Information] Dependency on non-open-source third-party contracts introduces unknown risks

Category: Unsafe External Call Audit

Content

In the `FundOperationLogic.sol` contract's `_PurchaseToSupplier` and `_RedemptionToSupplier` functions, which initiate purchase and redemption requests to suppliers, the system interacts with third-party contracts by calling their `subscribe` and `redeem` methods. However, these third-party contracts are outside the audit scope and their code is not publicly available, making it impossible to verify their internal logic, security posture, or potential vulnerabilities. This reliance introduces external risks such as possible fund manipulation or denial-of-service attacks.

Code location: contract/contracts/logic/FundOperationLogic.sol#L563-L596

```
    function _PurchaseToSupplier(
        uint256 purchaseCashAmount,
        FundState.Fund memory fund
    ) internal {
        ...
        ISupplier(supplierAddress).subscribe(
            fund.assetToken,
            fund.cashToken,
            purchaseCashAmount,
            block.timestamp + 1 days
        );
    }
```

```
    //向供应商发起赎回请求
    function _RedemptionToSupplier(
        uint256 redeemFundAmount,
        FundState.Fund memory fund
    ) internal {
        ...
        ISupplier(supplierAddress).redeem(
            fund.assetToken,
            fund.cashToken,
            redeemFundAmount,
            block.timestamp + 1 days
        );
    }
```

Solution

N/A

Status

Acknowledged

[N17] [Low] Potential risk due to missing reset step in approve operation

Category: Denial of Service Vulnerability

Content

In the FundOperationLogic.sol contract's `_PurchaseToSupplier` and `_RedemptionToSupplier` functions, which handle purchase and redemption requests to suppliers respectively, the code performs an `approve` operation by directly granting the target allowance (`purchaseCashAmount` or `redeemFundAmount`) without first resetting it to zero. This could lead to approval failures for certain tokens (such as USDT) that require the allowance to be set to `0` before a new value can be approved when residual allowances exist.

Code location: contract/contracts/logic/FundOperationLogic.sol#L563-L596

```
    function _PurchaseToSupplier(
        uint256 purchaseCashAmount,
        FundState.Fund memory fund
    ) internal {
        //向供应商发起申购请求
        IERC20(fund.cashToken).approve(
            supplierAddress,
            purchaseCashAmount + 10 ** IERC20Metadata(fund.cashToken).decimals()
```

```
        );
        ...
    }

    //向供应商发起赎回请求
    function _RedemptionToSupplier(
        uint256 redeemFundAmount,
        FundState.Fund memory fund
    ) internal {
        //向供应商发起赎回请求
        IERC20(fund.assetToken).approve(
            supplierAddress,
            redeemFundAmount + 10 ** IERC20Metadata(fund.assetToken).decimals()
        );
        ...
    }
```

Solution

Before approving the target amount, call `approve` with a `0` allowance to reset the existing authorization, and then perform a second `approve` to set the desired target amount.

Status

Acknowledged; After communicating with the project team, they confirmed that this is the intended design.

[N18] [Low] Excessive allowance to an unknown third-party contract causing risk exposure

Category: Authority Control Vulnerability Audit

Content

In the FundOperationLogic.sol contract's `_RedemptionToSupplier` function, which is responsible for initiating redemption requests to a supplier, the code performs an `approve` operation granting `redeemFundAmount` **plus** an additional `10**decimals` allowance to `supplierAddress`. Since this supplier is an external third-party contract with unknown risk, such excessive authorization could be maliciously exploited, leading to unintended fund exposure or loss.

Code location: contract/contracts/logic/FundOperationLogic.sol#L588

```
    function _RedemptionToSupplier(
        uint256 redeemFundAmount,
        FundState.Fund memory fund
    ) internal {
        //向供应商发起赎回请求
```

```
        IERC20(fund.assetToken).approve(
            supplierAddress,
            redeemFundAmount + 10 ** IERC20Metadata(fund.assetToken).decimals()
        );
        ...
    }
```

Solution

Remove the extra allowance and approve only the exact `redeemFundAmount`, adhering to the principle of least privilege.

Status

Acknowledged; After communicating with the project team, they confirmed that this is the intended design.

[N19] [Medium] Risk of excessive privilege

Category: Authority Control Vulnerability Audit

Content

In the `FOFToken.sol` contract's `burnFrom` function, the `minter` role is permitted to burn tokens from **any user address** as long as the balance is sufficient, without requiring additional authorization or constraints.

Code location: contract/contracts/tokens/FOFToken.sol#L121-L133

```
    function burnFrom(address from, uint256 amount)
        public
        override
        onlyMinter
        whenNotPaused
        nonReentrant
    {
        require(from != address(0), "FOFToken: burn from zero address");
        require(amount > 0, "FOFToken: burn amount must be greater than 0");
        require(balanceOf(from) >= amount, "FOFToken: burn amount exceeds balance");

        _burn(from, amount);
    }
```

Solution

Since the `minter` role is controlled by the `owner`, in the short term, transferring the privileged `owner` role to a multisignature wallet can effectively mitigate single-point-of-failure risks. In the long term, delegating these privileged

roles to a DAO governance mechanism would provide a sustainable solution to the issue of excessive privilege. During the transitional phase, managing the role via multisig combined with a timelock for delayed execution can further mitigate risks associated with excessive privilege.

Status

Acknowledged

[N20] [Suggestion] Centralized ownership privileges and lack of role separation

Category: Authority Control Vulnerability Audit

Content

In the FOFToken.sol contract's `pause` function, the `owner` role is granted the authority to pause the contract. However, since the `owner` also holds the management and administrative privileges, there is no separation of responsibilities, resulting in concentrated control.

Code location: contract/contracts/tokens/FOFToken.sol#L258

```
    function pause() external onlyOwner {
        _pause();
     }
```

Solution

Implement role separation by delegating the pause operation to an independent externally owned account (EOA) that can respond quickly to emergencies, while retaining the unpause authority for the `owner`.

Status

Acknowledged; After communicating with the project team, they confirmed that this is the intended design.

[N21] [Suggestion] Missing initialization state check during fund creation

Category: Design Logic Audit

Content

In the `FOFStorage.sol` contract's `createFund` function, which is responsible for creating new funds and storing their state, there is no verification of whether the input parameter `fund.isInitialized` is set to `true`. This

omission may result in uninitialized fund structures being passed in, preventing subsequent updates via `updateFund`, or allowing repeated `createFund` calls to create duplicate fund entries.

Code location: contract/contracts/storage/FOFStorage.sol#L77

```
    function createFund(string memory fundName, FundState.Fund memory fund) external
onlyAuthorized {
        require(!isFundExists(fundName), "FOFStorage: Fund already exists");
        funds[fundName] = fund;

        emit FundCreated(fundName, block.timestamp); // 使用时间戳作为唯一标识
    }
```

Solution

Add a validation step to check the `isInitialized` status of the input parameter to ensure the fund is properly initialized before creation.

Status

Acknowledged; After communicating with the project team, they confirmed that this is the intended design.

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002511060002	SlowMist Security Team	2025.11.03 - 2025.11.06	Low 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 1 high risk, 5 medium risk, 3 low risk, 7 suggestions, and 5 information. All the findings were fixed or acknowledged. The code was not deployed to the mainnet. The protocol still carries unresolved risks of excessive privilege. However, this is an inherent aspect of a semi-decentralized asset management design. Users should fully understand the protocol’s architecture and the associated risks.

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