Smart Contract Security Audit Report

Table Of Contents

1 Executive Summary

On 2024.07.10, the SlowMist security team received the SoSoValueLabs team's security audit application for SSI 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

SSI Protocol leverages on-chain smart contracts to repackage multi-chain, multi-asset portfolios into Wrapped Tokens. These tokens represent a basket of underlying assets, enabling Wrapped Tokens to track the value fluctuations of the basket.

3.2 Vulnerability Information

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

NO	Title	Category	Level	Status
N1	Missing event records	Others	Suggestion	Fixed

NO	Title	Category	Level	Status
N2	Missing scope limit	Others	Suggestion	Fixed
N3	Improper use of judgment symbols	Design Logic Audit	Medium	Fixed
N4	Redundant code	Others	Suggestion	Fixed
N5	Missing chain parameter check	Design Logic Audit	Information	Acknowledged
N6	Lack of amount check during rebalancing	Arithmetic Accuracy Deviation Vulnerability	Medium	Fixed
N7	Lack of Issue lock check	Design Logic Audit	Suggestion	Fixed
N8	Missing permission check	Authority Control Vulnerability Audit	Suggestion	Acknowledged
N9	Address conflict for funds transfer	Design Logic Audit	Information	Fixed
N10	Potential risk of denial of service due to large participants array	Gas Optimization Audit	Suggestion	Fixed
N11	Missing return value check	Others	Suggestion	Fixed
N12	Incorrect Tokenset comparison during addMintRequest and addRedeemRequest	Design Logic Audit	Information	Acknowledged
N13	Risk of excessive authority	Authority Control Vulnerability Audit	Medium	Acknowledged
N14	Missing ERC20 return value check	Others	Suggestion	Fixed
N15	Decimals loss issue	Arithmetic Accuracy Deviation Vulnerability	Information	Acknowledged

4 Code Overview

4.1 Contracts Description

Audit Version:

ssi-protocol.zip

SHA256: 8427c655116b4f34a6978b9e2971cf94299098098d8ba7bf75840207e6ebe31f

Fixed Version:

https://github.com/SoSoValueLabs/ssi-protocol

commit: 6058696c4e0dee930e12260144698d1a7c3a4b94

The main network address of the contract is as follows:

AssetFactory: https://arbiscan.io/address/0xbdfEE20D318Be5BD0b14a75CE94FB993fB2b587F

Swap: https://arbiscan.io/address/0xc34fD1d766f052f0A888badA893Bd95077652981

AssetIssuer: https://arbiscan.io/address/0x5f8e025a6Ac7144A9984005be7FAb035FF43a916

AssetFeeManager: https://arbiscan.io/address/0x98c7bEA94F953377285eBa454a638b4d46022FAD

AssetRebalancer: https://arbiscan.io/address/0x9ED7B49d25C29D03995e3cCEB25b528b456c9f05

4.2 Visibility Description

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

AssetController			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	Ownable

AssetFactory			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	Ownable
setSwap	External	Can Modify State	onlyOwner
setVault	External	Can Modify State	onlyOwner

AssetFactory			
createAssetToken	External	Can Modify State	onlyOwner
hasAssetID	External	-	-
getAssetIDs	External	-	-

AssetToken			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	ERC20
decimals	Public	-	-
getTokenset	Public	-	-
initTokenset	External	Can Modify State	onlyRole
setTokenset	Internal	Can Modify State	-
getBasket	Public	-	-
setBasket	Internal	Can Modify State	-
lockIssue	External	Can Modify State	onlyRole
issuing	External	-	-
unlockIssue	External	Can Modify State	onlyRole
mint	External	Can Modify State	onlyRole
burn	External	Can Modify State	onlyRole
lockRebalance	External	Can Modify State	onlyRole
unlockRebalance	External	Can Modify State	onlyRole
rebalance	External	Can Modify State	onlyRole
setFee	External	Can Modify State	onlyRole
getFeeTokenset	External	-	-

AssetToken

feeCollected	External	-	-
collectFeeTokenset	External	Can Modify State	onlyRole
lockBurnFee	External	Can Modify State	onlyRole
unlockBurnFee	External	Can Modify State	onlyRole
burnFeeTokenset	External	Can Modify State	onlyRole
setFeeTokenset	Internal	Can Modify State	-

AssetRebalancer

Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	AssetController
getRebalanceRequestLength	External	-	-
getRebalanceRequest	External	-	-
addRebalanceRequest	External	Can Modify State	onlyOwner
rejectRebalanceRequest	External	Can Modify State	onlyOwner
confirmRebalanceRequest	External	Can Modify State	onlyOwner

AssetFeeManager

Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	AssetController
setFee	External	Can Modify State	onlyOwner
collectFeeTokenset	External	Can Modify State	onlyOwner
getBurnFeeRequestLength	External	-	-
getBurnFeeRequest	External	-	-

AssetFeeManager			
addBurnFeeRequest	External	Can Modify State	onlyOwner
rejectBurnFeeRequest	External	Can Modify State	onlyOwner
confirmBurnFeeRequest	External	Can Modify State	onlyOwner

AssetIssuer			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	AssetController
pause	External	Can Modify State	onlyOwner
unpause	External	Can Modify State	onlyOwner
getIssueAmountRange	External	-	-
setIssueAmountRange	External	Can Modify State	onlyOwner
getIssueFee	External	-	-
setIssueFee	External	Can Modify State	onlyOwner
getMintRequestLength	External	-	-
getMintRequest	External	-	-
addMintRequest	External	Can Modify State	-
rejectMintRequest	External	Can Modify State	onlyOwner
confirmMintRequest	External	Can Modify State	onlyOwner
getRedeemRequestLength	External	-	-
getRedeemRequest	External	-	-
addRedeemRequest	External	Can Modify State	-
rejectRedeemRequest	External	Can Modify State	onlyOwner
confirmRedeemRequest	External	Can Modify State	onlyOwner

AssetIssuer			
isParticipant	External	-	-
getParticipants	External	-	-
addParticipant	External	Can Modify State	onlyOwner
removeParticipant	External	Can Modify State	onlyOwner

Swap			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
checkOrderInfo	Public	-	-
getOrderHashs	External	-	-
addSwapRequest	External	Can Modify State	onlyRole
getSwapRequest	External	-	-
makerRejectSwapRequest	External	Can Modify State	onlyRole
makerConfirmSwapRequest	External	Can Modify State	onlyRole
rollbackSwapRequest	External	Can Modify State	onlyRole
confirmSwapRequest	External	Can Modify State	onlyRole
setTakerAddresses	External	Can Modify State	onlyRole
getTakerAddresses	External	-	-

4.3 Vulnerability Summary

[N1] [Suggestion] Missing event records

Category: Others

Content

In the AssetFactory contract, the owner role can modify the swap and vault addresses through the setSwap and setVault functions, respectively. However, no event logging is performed.

Code Location:

src/AssetFactory.sol#L29-37

```
    function setSwap(address swap_) external onlyOwner {
        require(swap_ != address(0), "swap address is zero");
        swap = swap_;
    }

    function setVault(address vault_) external onlyOwner {
        require(vault_ != address(0), "vault address is zero");
        vault = vault_;
    }
```

Solution

It is recommended to implement event logging when modifying sensitive parameters to facilitate subsequent self-inspection or community auditing.

Status

Fixed

[N2] [Suggestion] Missing scope limit

Category: Others

Content

1.In the AssetFactory contract, the owner role can call the createAssetToken function to create a new AssetToken. When initializing AssetToken, the maxFee parameter will be passed to set the maximum fee limit. However, there is a lack of range restrictions on maxFee, if the set numeric value is too large (for example, more than 1e8), then the set fee may also exceed 1e8, which may cause unexpected consumption of user funds.

Code Location:

src/AssetFactory.sol#L46

```
    function createAssetToken(Asset memory asset, uint maxFee, address issuer,
address rebalancer, address feeManager) external onlyOwner returns (address) {
        ...
        AssetToken assetToken = new AssetToken({
            id_: asset.id,
            name_: asset.name,
            symbol_: asset.symbol,
            maxFee_: maxFee,
            owner: address(this)
        });

        ...
    }
```

2.In the AssetIssuer contract, the owner role can call the setIssueFee function to set the issueFee. However, there is a lack of range restrictions on issueFee, if the set numeric value is too large, which may cause unexpected consumption of user funds.

Code Location:

src/AssetIssuer.sol#L76

```
    function setIssueFee(uint256 assetID, uint256 issueFee) external onlyOwner {
        _issueFees.set(assetID, issueFee);
        emit SetIssueFee(assetID, _issueFees.get(assetID));
    }
```

Solution

It is recommended to add the range checking to the corresponding functions.

Status

Fixed

[N3] [Medium] Improper use of judgment symbols

Category: Design Logic Audit

Content

In the AssetToken contract, the fee manager contract can collect the daily fee by calling the collectFeeTokenset function. However, the check in function is that the fee can only be collected when the difference between the current timestamp and the last collection time (lastCollectTimestamp) is greater than 1 day.

If the difference between the current timestamp and the last collection time (lastCollectTimestamp) is exactly 1 day but the fee cannot be collected, this does not meet normal design expectations.

Code Location:

src/AssetToken.sol#L152

```
    function collectFeeTokenset() external onlyRole(FEEMANAGER_ROLE) {
        if (block.timestamp - lastCollectTimestamp > 1 days) {

            ...

        }
    }
```

Solution

It is recommended to change the greater than 1 day in the judgment to greater than or equal to 1 day.

Status

Fixed

[N4] [Suggestion] Redundant code

Category: Others

Content

In the AssetToken contract, the fee manager contract can collect the daily fee by calling the collectFeeTokenset function. However, in this function, the time to collect the fee is repeatedly checked, which may consume additional gas.

Code Location:

src/AssetToken.sol#L158

```
    function collectFeeTokenset() external onlyRole(FEEMANAGER_ROLE) {
        if (block.timestamp - lastCollectTimestamp > 1 days) {
            if (totalSupply() > 0) {
                ...

                uint256 feeDays = (block.timestamp - lastCollectTimestamp) / 1 days;
                if (feeDays > 0) {
                    ...
                    }
                }
            }
        }
```

Solution

It is recommended to remove the redundant if judgment on the feeDays parameter.

Status

Fixed

[N5] [Information] Missing chain parameter check

Category: Design Logic Audit

Content

In the AssetRebalancer contract, the owner role can commit a rebalancing operation request by calling the addRebalanceRequest function. As expected by design, the rebalancing operation may add new underlying tokens to AssetToken. However, there is no check for order.outTokenset here.

Code Location:

src/AssetRebalancer.sol

```
    function addRebalanceRequest(uint256 assetID, Token[] memory basket, OrderInfo
memory orderInfo) external onlyOwner returns (uint256) {
        ...
    }

    ...

    function confirmRebalanceRequest(uint nonce, bytes32[] memory inTxHashs) external
onlyOwner {
        ...

        Order memory order = swapRequest.orderInfo.order;
        Token[] memory inBasket = Utils.muldivTokenset(order.outTokenset,
order.outAmount, 10**8);

        ...
    }
```

Solution

N/A

Status

Acknowledged; The project team's response: rebalance will change assetToken.tokenset, but mint only check chain of order.inTokenset and redemption only check chain of order.outTokenset, which is not consistent with assetToken.tokenset. It won't cause issue functions fail to work after rebalancing.

Final determination: This is not an issue.

[N6] [Medium] Lack of amount check during rebalancing

Category: Arithmetic Accuracy Deviation Vulnerability

Content

In the AssetToken contract, the AssetRebalancer contract can call the rebalance function to update the basket and tokenset. When calculating a new tokenset, due to the lack of checking in the AssetRebalancer contract whether the amount of each outToken token multiplied by the value of order.outAmount is not less than AssetToken.totalSupply(), it may cause the amount of the newly added currency in the new tokenset to be calculated as 0.

Code Location:

src/AssetRebalancer.sol

```
    function confirmRebalanceRequest(uint nonce, bytes32[] memory inTxHashs) external
onlyOwner {
        ...

        Order memory order = swapRequest.orderInfo.order;
        Token[] memory inBasket = Utils.muldivTokenset(order.outTokenset,
order.outAmount, 10**8);

        ...
    }
```

src/AssetToken.sol#L129

```
    function rebalance(Token[] memory inBasket, Token[] memory outBasket) external
onlyRole(REBALANCER_ROLE) {
        require(rebalancing, "lock rebalance first");
        Token[] memory newBasket = Utils.addTokenset(Utils.subTokenset(basket_,
outBasket), inBasket);
        Token[] memory newTokenset = Utils.muldivTokenset(newBasket, 10**decimals(),
totalSupply());
        setBasket(newBasket);
        setTokenset(newTokenset);
    }
```

Solution

It is recommended to add a check to the AssetRebalancer contract that the amount of each outToken token multiplied by the value of order.outAmount cannot be less than AssetToken.totalSupply().

Status

Fixed

[N7] [Suggestion] Lack of Issue lock check

Category: Design Logic Audit

Content

In the AssetFeeManager contract, the owner can call the collectFeeTokenset function to collect the daily fee. However, in this function only the rebalance lock is checked and not the issue lock.

Code Location:

src/AssetFeeManager.sol#L29-35

```
    function collectFeeTokenset(uint256 assetID) external onlyOwner {
        IAssetFactory factory = IAssetFactory(factoryAddress);
        IAssetToken assetToken = IAssetToken(factory.assetTokens(assetID));
        require(assetToken.hasRole(assetToken.FEEMANAGER_ROLE(), address(this)), "not
a fee manager");
        require(assetToken.rebalancing() == false, "is rebalancing");
        assetToken.collectFeeTokenset();
    }
```

Solution

It is recommended to add a check for issue lock in the collectFeeTokenset function.

Status

Fixed

[N8] [Suggestion] Missing permission check

Category: Authority Control Vulnerability Audit

Content

In the AssetIssuer contract, the participant roles corresponding to the specified AssetToken can call the addMintRequest function to add a minting request for the AssetToken token.

However, in this function, there is a lack of checks on whether this contract has issuer role permissions. In both AssetRebalancer and AssetFeeManager contracts, contract permissions are checked when committing new requests.

Code Location:

src/AssetIssuer.sol#L89-134

```
    function addMintRequest(uint256 assetID, OrderInfo memory orderInfo) external
returns (uint) {
    ...
}
```

Solution

It is recommended to add a check on whether this contract has issuer role privilege to the addMintRequest function.

Status

Acknowledged; The project team's response: It's by design, because participants need to invoke this function, and the contract checks the privilege role by assetID inside the function.

[N9] [Information] Address conflict for funds transfer

Category: Design Logic Audit

Content

In the AssetIssuer contract, the participant roles corresponding to the specified AssetToken can call the addMintRequest function to add a minting request for the AssetToken token. And the owner role can call the rejectMintRequest function to reject the minting request and return the token transferred when the request was previously committed to the participant role.

When committing a minting request, the addMintRequest function transfers the underlying token into the vault contract. However, when rejecting the request and redeeming, it directly returns the token from the AssetIssuer contract to the user.

Code Location:

src/AssetIssuer.sol

```
    function addMintRequest(uint256 assetID, OrderInfo memory orderInfo) external
returns (uint) {
        ...

        for (uint i = 0; i < inTokenset.length; i++) {
            require(bytes32(bytes(inTokenset[i].chain)) ==
bytes32(bytes(factory.chain())), "chain not match");
            address tokenAddress = Utils.stringToAddress(inTokenset[i].addr);
            IERC20 inToken = IERC20(tokenAddress);
            uint inTokenAmount = inTokenset[i].amount * order.inAmount / 10**8;
            uint feeTokenAmount = inTokenAmount * issueFee / 10**feeDecimals;
            uint transferAmount = inTokenAmount + feeTokenAmount;
            require(inToken.balanceOf(msg.sender) >= transferAmount, "not enough
balance");
            require(inToken.allowance(msg.sender, address(this)) >= transferAmount,
"not enough allowance");
            inToken.transferFrom(msg.sender, vault, transferAmount);
        }

        ...
    }

    ...

    function rejectMintRequest(uint nonce) external onlyOwner {
        ...

            inToken.transfer(mintRequest.requester, transferAmount);
        }

        ...
    }

    ...

    function confirmRedeemRequest(uint nonce, bytes32[] memory inTxHashs) external
onlyOwner {
        ...

        for (uint i = 0; i < outTokenset.length; i++) {
            address tokenAddress = Utils.stringToAddress(outTokenset[i].addr);
            IERC20 outToken = IERC20(tokenAddress);
            uint outTokenAmount = outTokenset[i].amount * order.outAmount / 10**8;
            uint feeTokenAmount = outTokenAmount * redeemRequest.issueFee /
10**feeDecimals;
            uint transferAmount = outTokenAmount - feeTokenAmount;
            require(outToken.balanceOf(address(this)) >= transferAmount, "not enough
balance");
            outToken.transfer(redeemRequest.requester, transferAmount);
        }

        ...
    }
```

Solution

It is recommended that the address that funds are transferred to when minting tokens is the same as the source address of funds when refunding.

Status

Fixed; The project team's response: during minting, order maker only trades with vault wallet, therefore the funds need to be transferred to vault wallet. Before rejecting the mint request, vault wallet will transfer funds back to the contract.

Final conclusion: This conforms to the project team's anticipated design and is not a problem.

[N10] [Suggestion] Potential risk of denial of service due to large participants array

Category: Gas Optimization Audit

Content

In the AssetIssuer contract, when the owner role performs getParticipants operation, it will use a for loop to traverse the entire participants array. If the array length is too large, it will lead to DoS risks.

Code Location:

src/AssetIssuer.sol#L274-285

```
    function getParticipants(uint256 assetID) external view returns (address[]
memory) {
        address[] memory participants = new address[]
(_participants[assetID].length());
        for (uint i = 0; i < participants.length; i++) {
            participants[i] = _participants[assetID].at(i);
        }
        return participants;
    }

    function addParticipant(uint256 assetID, address participant) external onlyOwner
{
        _participants[assetID].add(participant);
        emit AddParticipant(assetID, participant);
    }
```

Solution

It is recommended to set the quantity cap when calling the addParticipant function to add data to the participants array.

Status

Fixed; The project team designed a new function to obtain the specified data in the array.

[N11] [Suggestion] Missing return value check

Category: Others

Content

In the AssetIssuer contract, the owner role can call addParticipant and removeParticipant to add or remove data from the participant array. When using OpenZeppelin's EnumerableMap library, the .add() and .remove() functions will return a boolean value, but there is no check to see if the return value is true.

Code Location:

src/AssetIssuer.sol#L282-290

```
    function addParticipant(uint256 assetID, address participant) external onlyOwner
{
        _participants[assetID].add(participant);
        emit AddParticipant(assetID, participant);
    }

    function removeParticipant(uint256 assetID, address participant) external
onlyOwner {
        _participants[assetID].remove(participant);
        emit RemoveParticipant(assetID, participant);
    }
```

Solution

It is recommended to add a check to see if the return value of the add and remove functions is true.

Status

Fixed

[N12] [Information] Incorrect Tokenset comparison during addMintRequest and addRedeemRequest

Category: Design Logic Audit

Content

In the AssetIssuer contract, the owner role can commit requests to mint and redeem AssetTokens by calling the addMintRequest and addRedeemRequest functions.

However, when committing a mint request, hash consistency is checked with order.outTokenset and assetToken.getTokenset (). The relevant data transferred by the underlying token is stored in order.inTokenset. If order.inTokenset is different from assetToken.getTokenset (), it may lead to unexpected errors that cause loss in capital. The same problem occurs when committing a redemption request, and consistency checks with order.inTokenset and assetToken.getTokenset () should not be used.

Code Location:

src/AssetIssuer.sol#L104&L202

```
    function addMintRequest(uint256 assetID, OrderInfo memory orderInfo) external
returns (uint) {
        ...

        require(keccak256(abi.encode(assetToken.getTokenset())) ==
keccak256(abi.encode(order.outTokenset)), "tokenset not match");

        ...
    }

    ...

    function addRedeemRequest(uint256 assetID, OrderInfo memory orderInfo) external
returns (uint256) {
        ...

        require(keccak256(abi.encode(assetToken.getTokenset())) ==
keccak256(abi.encode(order.inTokenset)), "tokenset not match");
```

```
        ...
    }
```

Solution

N/A

Status

Acknowledged; The project team's response: when committing mint requests:

1.participant pays order.inTokenset to buy order.outTokenset.

2.AssetIssuer contract transfers order.inTokenset to vault wallet, and request order maker to fullfill the order.

3.Once order maker transfer order.outTokenset to vault wallet, vault wallet will transfer order.inTokenset to maker

4.AssetIssuer owner confirm mintRequest and mint AssetToken to participant

Therefore AssetToken.tokenset should be consistent to order.outTokenset.

Final conclusion: This conforms to the project team's anticipated design and is not a problem.

[N13] [Medium] Risk of excessive authority

Category: Authority Control Vulnerability Audit

Content

1.In the AssetIssuer contract, the owner role can modify the configuration of assetToken, such as the _minAmounts, the maxAmounts and the _issueFees. And the owner is responsible for rejecting and confirming the committed mint and redemption requests. If the private key of the owner role is leaked, it will cause the loss of contract funds.

Code Location:

src/AssetIssuer.sol

```
    function setIssueAmountRange(uint256 assetID, Range calldata issueAmountRange)
external onlyOwner {
        ...
    }

    function setIssueFee(uint256 assetID, uint256 issueFee) external onlyOwner {
       ...
    }

    ...
    function rejectMintRequest(uint nonce) external onlyOwner {
        ...
    }

    function confirmMintRequest(uint nonce, bytes32[] memory inTxHashs) external
onlyOwner {
        ...
    }

    ...
    function rejectRedeemRequest(uint nonce) external onlyOwner {
        ...
    }

    function confirmRedeemRequest(uint nonce, bytes32[] memory inTxHashs) external
onlyOwner {
        ...
    }
```

2.In the AssetRebalancer contract, the owner role can modify the underlying token data of AssetToken through the addRebalanceRequest function. If the private key of the owner role is leaked, it may cause the data of the underlying token in AssetToken to be arbitrarily tampered with and lose its original value.

Code Location:

src/AssetRebalancer.sol

```
    function addRebalanceRequest(uint256 assetID, Token[] memory basket, OrderInfo
memory orderInfo) external onlyOwner returns (uint256) {
        ...
    }

    function rejectRebalanceRequest(uint nonce) external onlyOwner {
        ...
    }

    function confirmRebalanceRequest(uint nonce, bytes32[] memory inTxHashs) external
onlyOwner {
        ...
    }
```

3.In the Swap contrac, the maker role has permissions to sign, reject, and confirm the order data. The taker role has permissions to add, roll back, and reconfirm the order data. If the private key of these roles is leaked, it will cause the loss of contract funds.

Code Location:

src/Swap.sol

```
    function makerRejectSwapRequest(bytes32 orderHash) external onlyRole(MAKER_ROLE)
{
        ...
    }

    function makerConfirmSwapRequest(bytes32 orderHash, bytes32[] memory outTxHashs)
external onlyRole(MAKER_ROLE) {
        ...
    }

    function rollbackSwapRequest(bytes32 orderHash) external onlyRole(TAKER_ROLE) {
        ...
    }

    function confirmSwapRequest(bytes32 orderHash, bytes32[] memory inTxHashs)
external onlyRole(TAKER_ROLE) {
        ...
    }
```

Solution

In the short term, transferring the ownership of core roles 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. Like the authority involving user funds should be managed by the community.

Status

Acknowledged; The project team's response: will use cobo mpc custody wallets to manage the ownership of the contracts. The owners of AssetIusser/Rebalancer/FeeManager can be different wallets, and AssetToken has three different privileged roles.

Update: The contract was created using Cobo's MPC wallet, so the permissions for the core roles have been transferred to the MPC wallet for management. This has also been confirmed via official email by Cobo. The following are the transactions made during the setting of permissions:

https://arbiscan.io/tx/0x50e84b4425f1e19da963999aa73d8240f6bb1fb8ab75125f8aab5bac372dacaf

https://arbiscan.io/tx/0xe23e4c6a36847c5df3ca1e7c809579c8965bab617e478da9ee376595fb33bd90

https://arbiscan.io/tx/0x596f6ca527fc973bc56fe14080bc7da2de3803edd3ca2aea2b78197e599fcbd7

https://arbiscan.io/tx/0x8cb2fec77aa30fe9bb2c80c0228a404dfa73cc9efaf8299df1ad068913a9a130

https://arbiscan.io/tx/0xdb658cf361d8066168e72472f2e68c7902f0d7bc7102feca9db0cbca98cab911

The taker role for the Swap contract has been set to the relevant contracts; the following are the transactions for this setting:

https://arbiscan.io/tx/0x68f68e934b62293a5ec158f3b7505695e67e4cd3c6761d121c410542be66d7f9

https://arbiscan.io/tx/0x3819f17aeee1af5f405755fbfc9eddcf8a3f156e56c6cd9789997970d13a053c

https://arbiscan.io/tx/0xb0ebce63966caa20ff40be2dabcd2ecaac4f4606dc29f025b34a07144cfde31c

When the maker role confirms the swap in the Swap contract, it's just for bookkeeping purposes; only when the taker role confirms will the user's funds be transferred to the maker. If the maker unexpectedly confirms, the taker role will roll back the confirm status of the maker role. Additionally, before the owner role confirms the transaction, they will check if the outTxHash meets expectations:

- Whether the amount transferred to the vault is as expected.
- Whether the orderHash is included in the data field of the transfer transaction.

[N14] [Suggestion] Missing ERC20 return value check

Category: Others

Content

In the AssetIssuer contract, the user can call function to transfer tokens in the contract. But it does not check the return value. If external tokens do not adopt the EIP20 standard, it may lead to “false top-up” issues.

Code Location:

src/AssetIssuer.sol

```
    function addMintRequest(uint256 assetID, OrderInfo memory orderInfo) external
returns (uint) {
        ...
        for (uint i = 0; i < inTokenset.length; i++) {
            ...
            inToken.transferFrom(msg.sender, vault, transferAmount);
        }
        ...
    }

    function rejectMintRequest(uint nonce) external onlyOwner {
        ...
        for (uint i = 0; i < inTokenset.length; i++) {
            ...
            inToken.transfer(mintRequest.requester, transferAmount);
        }
        ...
    }

    function addRedeemRequest(uint256 assetID, OrderInfo memory orderInfo) external
returns (uint256) {
        ...
        for (uint i = 0; i < outTokenset.length; i++) {
            require(bytes32(bytes(outTokenset[i].chain)) ==
bytes32(bytes(factory.chain())), "chain not match");
        }
        assetToken.transferFrom(msg.sender, address(this), order.inAmount);
        ...
    }

    function rejectRedeemRequest(uint nonce) external onlyOwner {
        ...
        assetToken.transfer(redeemRequest.requester, redeemRequest.amount);
        ...
    }

    function confirmRedeemRequest(uint nonce, bytes32[] memory inTxHashs) external
onlyOwner {
        ...
        for (uint i = 0; i < outTokenset.length; i++) {
            ...
            outToken.transfer(redeemRequest.requester, transferAmount);
        }
        ...
    }
```

Solution

It is recommended to add a check of the return value or use SafeERC20 library.

Status

Fixed

[N15] [Information] Decimals loss issue

Category: Arithmetic Accuracy Deviation Vulnerability

Content

The following function in AssetIssuer contract may encounter a decimals loss issue when calculating inTokenAmount and feeTokenAmount. If the decimal of order.inAmount is not 1e8 but represents the decimal of the token itself and when the decimal of the underlying token represented by inTokenset [i] is less than 1e8 (e.g. the decimal of USDT is 1e6), then the calculated inTokenAmount may differ many times from what is actually expected because the divisor is fixed at 1e8.

Code Location:

src/AssetIssuer.sol

```
    function addMintRequest(uint256 assetID, OrderInfo memory orderInfo) external
returns (uint) {
        ...
        for (uint i = 0; i < inTokenset.length; i++) {
            require(bytes32(bytes(inTokenset[i].chain)) ==
bytes32(bytes(factory.chain())), "chain not match");
            address tokenAddress = Utils.stringToAddress(inTokenset[i].addr);
            IERC20 inToken = IERC20(tokenAddress);
            uint inTokenAmount = inTokenset[i].amount * order.inAmount / 10**8;
            uint feeTokenAmount = inTokenAmount * issueFee / 10**feeDecimals;
            uint transferAmount = inTokenAmount + feeTokenAmount;
            require(inToken.balanceOf(msg.sender) >= transferAmount, "not enough
balance");
            require(inToken.allowance(msg.sender, address(this)) >= transferAmount,
"not enough allowance");
            inToken.transferFrom(msg.sender, vault, transferAmount);
        }
        ...
    }


    ...

     function rejectMintRequest(uint nonce) external onlyOwner {
         ...

         for (uint i = 0; i < inTokenset.length; i++) {
             require(bytes32(bytes(inTokenset[i].chain)) ==
bytes32(bytes(factory.chain())), "chain not match");
```

```
            address tokenAddress = Utils.stringToAddress(inTokenset[i].addr);
            IERC20 inToken = IERC20(tokenAddress);
            uint inTokenAmount = inTokenset[i].amount * order.inAmount / 10**8;
            uint feeTokenAmount = inTokenAmount * mintRequest.issueFee /
10**feeDecimals;
            uint transferAmount = inTokenAmount + feeTokenAmount;
            require(inToken.balanceOf(address(this)) >= transferAmount, "not enough
balance");
            inToken.transfer(mintRequest.requester, transferAmount);
        }

        ...
    }


   function confirmRedeemRequest(uint nonce, bytes32[] memory inTxHashs) external
onlyOwner {
       ...
       for (uint i = 0; i < outTokenset.length; i++) {
           address tokenAddress = Utils.stringToAddress(outTokenset[i].addr);
           IERC20 outToken = IERC20(tokenAddress);
           uint outTokenAmount = outTokenset[i].amount * order.outAmount / 10**8;
           uint feeTokenAmount = outTokenAmount * redeemRequest.issueFee /
10**feeDecimals;
           uint transferAmount = outTokenAmount - feeTokenAmount;
           require(outToken.balanceOf(address(this)) >= transferAmount, "not enough
balance");
           outToken.transfer(redeemRequest.requester, transferAmount);
       }
       ...
   }
```

Solution

If the decimal of order.inAmount is not 1e8 but represents the decimal of the token itself, it is recommended to use the accuracy corresponding to the underlying token as a divisor instead of a fixed 1e8 when calculating inTokenAmount, and it performs multiplication first and includes rounding factors before performing division.

Status

Acknowledged; The project team's response: by design, the decimals of order.inAmount and order.outAmount is 1e8.

Final conclusion: This conforms to the project team's anticipated design and is not a problem.

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002407150002	SlowMist Security Team	2024.07.10 - 2024.07.15	Low 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 3 medium risk, 8 suggestion vulnerabilities and 4 information. All the findings were fixed and acknowledged. The project has been deployed on the mainnet, and the permissions of the core roles have been transferred to be managed by the Cobo's MPC wallet and the contracts.

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.