Smart Contract Security Audit Report

Table Of Contents

1 Executive Summary

On 2025.11.10, the SlowMist security team received the USiC Labs team's security audit application for USiC, 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

USiC is a decentralized interest-bearing stablecoin protocol based on Ethereum. Users can mint USiC tokens at a 1:1 ratio by depositing USDC or USDT, and automatically earn a fixed APY (Annual Percentage Yield) linked to the Federal Reserve's benchmark interest rate (approximately 4.25%), achieving continuous compound interest without manual operation. The protocol adopts a Transparent Proxy upgradable architecture. The funds are mainly invested in low-risk assets, with 80% temporarily stored through Aave and then transferred to US Treasury bonds and high-rated corporate bonds, and 20% deposited into Yearn V3 to optimize DeFi returns. The comprehensive yield is 4-8%, and the interest rate spread serves as the protocol's profit. Core functions include minting, redemption, transfer, batch operations, and redemption requests. It supports a 7-day lock-up period, sanction checks, account freezing, re-entry protection, and a pause mechanism to ensure security and liquidity. The project uses the OpenZeppelin

library and ABDKMath64x64 for high-precision calculations, and combines the Chainlink oracle for USDT de-peg detection. The overall design emphasizes modularity, access control, and a hybrid strategy of traditional finance and DeFi.

3.2 Vulnerability Information

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

NO	Title	Category	Level	Status
N1	Interest Minting Fails to Update Total Minted Amount Statistics	Design Logic Audit	Information	Acknowledged
N2	Potential Hard-Coded Precision Conversion Issues	Arithmetic Accuracy Deviation Vulnerability	Medium	Fixed
N3	Lack of Order Verification in Fee Threshold Update	Design Logic Audit	Low	Fixed
N4	Redundant Code	Others	Suggestion	Fixed
N5	Lack of Automatic Unfreezing in Beneficiary Freeze Check	Design Logic Audit	Low	Fixed
N6	Lack of Update to the Holder Set	Design Logic Audit	Suggestion	Fixed
N7	Lack of Check for Paused Redemption	Design Logic Audit	Medium	Fixed
N8	Lack of Verification of User Freeze and Sanction Status in the batchFulfillRedemption Function	Design Logic Audit	Medium	Fixed
N9	Optimizable Batch Redemption Amount Check	Gas Optimization Audit	Suggestion	Acknowledged

NO	Title	Category	Level	Status
N10	Gas Optimization in the allocateFunds Function	Gas Optimization Audit	Suggestion	Fixed
N11	Risk of excessive privilege	Authority Control Vulnerability Audit	Medium	Acknowledged
N12	Potential Bank Run Situation	Others	Information	Acknowledged
N13	Potential External Protocol Risks	Others	Information	Acknowledged

4 Code Overview

4.1 Contracts Description

Audit Version:

https://github.com/usic-labs/usic-core

commit: fa75211eab55ab706adabd9d76b07a63dc2ea9a2

Fixed Version:

https://github.com/usic-labs/usic-core

commit: 875a61c6e0c587d363b50f645f92d1c305f6720a

Audit Scope:

```
./contracts
├── USICBase.sol
├── USICCoreAdmin.sol
├── USICCoreUpgradeable.sol
├── USICDeFiUpgradeable.sol
├── USICStorage.sol
└── libraries
    └── ABDKMath64x64.sol
```

The main network address of the contract is as follows:

USiCCore: https://etherscan.io/address/0xcd6b66E917f6AFA7069833E895D5Fd94aaa43A35

USiCDeFi: https://etherscan.io/address/0x8580f2F9f713aa0d7ed80c6f63F2F6DBe38185FA

USICCoreAdmin: https://etherscan.io/address/0xFb1567212fF25F89F6A5D7C7F17fC6b15F217851

4.2 Visibility Description

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

USICBase			
Function Name	Visibility	Mutability	Modifiers
_updateGlobalGrowthFactor	Internal	Can Modify State	-
_updateUserGrowthFactor	Internal	Can Modify State	-
_isAccountFrozen	Internal	-	-
_checkAndAutoUnfreeze	Internal	Can Modify State	-
isSanctioned	Public	-	-
calculateRedeemableUSDC	Public	-	-
calculateRedemptionFee	Public	-	-
getAvailableBalance	Public	-	-
getTransferableBalance	Public	-	-
getLockedAmount	Public	-	-
balanceOf	Public	-	-

USICCoreAdmin			
Function Name	Visibility	Mutability	Modifiers
_setInterestRate	External	Can Modify State	-
_setMaxInterestRate	External	Can Modify State	-
_setMintLimits	External	Can Modify State	-

USICCoreAdmin			
_setLockPeriod	External	Can Modify State	-
_setFeeConfig	External	Can Modify State	-
_setAllFeeConfigs	External	Can Modify State	-
_setDeFiContract	External	Can Modify State	-
_setSanctionsList	External	Can Modify State	-
_setMintStatus	External	Can Modify State	-
_setRedeemPaused	External	Can Modify State	-
_freezeAccount	External	Can Modify State	-
_unfreezeAccount	External	Can Modify State	-
_pauseContract	External	Can Modify State	-
_unpauseContract	External	Can Modify State	-
_setAdminModule	External	Can Modify State	-
version	Public	-	-

USICCoreUpgradeable			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
initialize	Public	Can Modify State	initializer
version	Public	-	-
mint	External	Can Modify State	nonReentrant whenNotPaused whenAccountNotFrozen whenNotSanctioned updateUserInterest
redeem	External	Can Modify State	nonReentrant whenNotPaused whenRedeemNotPaused

USICCoreUpgradeable			
			whenAccountNotFrozen whenNotSanctioned updateUserInterest
requestRedemption	External	Can Modify State	nonReentrant whenNotPaused whenAccountNotFrozen whenNotSanctioned updateUserInterest
cancelRedemptionRequest	External	Can Modify State	nonReentrant whenNotPaused whenAccountNotFrozen updateUserInterest
transfer	Public	Can Modify State	whenNotPaused whenAccountNotFrozen whenAccountNotFrozen whenNotSanctioned whenNotSanctioned updateUserInterest updateUserInterest
transferFrom	Public	Can Modify State	whenNotPaused whenAccountNotFrozen whenAccountNotFrozen whenNotSanctioned whenNotSanctioned updateUserInterest updateUserInterest
setAdminModule	External	Can Modify State	onlyRole
setInterestRate	External	Can Modify State	onlyRole updateInterest
setMaxInterestRate	External	Can Modify State	onlyRole
setMintLimits	External	Can Modify State	onlyRole
setLockPeriod	External	Can Modify State	onlyRole
setFeeConfig	External	Can Modify State	onlyRole
setAllFeeConfigs	External	Can Modify State	onlyRole
setDeFiContract	External	Can Modify State	onlyRole
setSanctionsList	External	Can Modify State	onlyRole
setMintStatus	External	Can Modify State	onlyRole
setRedeemPaused	External	Can Modify State	onlyRole

USICCoreUpgradeable			
freezeAccount	External	Can Modify State	onlyRole
unfreezeAccount	External	Can Modify State	onlyRole
pause	External	Can Modify State	onlyRole
unpause	External	Can Modify State	onlyRole
emergencyWithdraw	External	Can Modify State	onlyRole whenPaused nonReentrant
depositUSDC	External	Can Modify State	onlyRole nonReentrant
batchFulfillRedemption	External	Can Modify State	onlyRole nonReentrant updateBatchUserInterest
batchMint	External	Can Modify State	onlyRole nonReentrant whenNotPaused updateBatchUserInterest
getUserMintHistory	External	-	-
getUserReceiveHistory	External	-	-
getCurrentGrowthFactor	External	-	-
getUserGrowthFactor	External	-	-
getAllFeeConfigs	External	-	-
isAccountFrozen	External	-	-
getUSICHoldersCount	External	-	-
isUSICHolder	External	-	-
getTotalHoldersBalance	External	-	-
getYearnInvestmentStatus	External	-	-
getYearnVaultInfo	External	-	-

USICCoreUpgradeable			
getOTCWalletAave Balance	External	-	-

USICDeFiUpgradeable			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
initialize	Public	Can Modify State	initializer
version	Public	-	-
allocateFunds	External	Can Modify State	onlyCore nonReentrant
swapUSDTToUSDC	External	Can Modify State	onlyCore nonReentrant
withdrawFromYearn	External	Can Modify State	onlyCore nonReentrant
depositRemainingToYearn	External	Can Modify State	onlyCore nonReentrant
_depositToYearn	Internal	Can Modify State	-
_withdrawFromYearn	Internal	Can Modify State	-
_swapUSDTToUSDC	Internal	Can Modify State	-
isUSDTWithinPeg	Public	-	-
setCoreContract	External	Can Modify State	onlyRole
setOTCWallet	External	Can Modify State	onlyRole
setYearnVault	External	Can Modify State	onlyRole nonReentrant
setCurve3Pool	External	Can Modify State	onlyRole

USICDeFiUpgradeable			
setAavePool	External	Can Modify State	onlyRole
setAaveAllocationPercent	External	Can Modify State	onlyRole
setMaxSlippage	External	Can Modify State	onlyRole
setMaxLossOnRedeem	External	Can Modify State	onlyRole
setUSDTPriceFeed	External	Can Modify State	onlyRole
setDepegThreshold	External	Can Modify State	onlyRole
setPriceFreshnessThreshold	External	Can Modify State	onlyRole
depositToYearn	External	Can Modify State	onlyRole nonReentrant
adminWithdrawFromYearn	External	Can Modify State	onlyRole nonReentrant
pause	External	Can Modify State	onlyRole
unpause	External	Can Modify State	onlyRole
emergencyWithdraw	External	Can Modify State	onlyRole whenPaused nonReentrant
getYearnInvestmentStatus	External	-	-
getYearnVaultInfo	External	-	-
getOTCWalletAaveBalance	External	-	-

4.3 Vulnerability Summary

[N1] [Information] Interest Minting Fails to Update Total Minted Amount Statistics

Category: Design Logic Audit

Content

In the _updateUserGrowthFactor function of USICBase.sol, this function is responsible for updating the user growth factor and calculating interest. When the interest is calculated (newBalance > currentBalance), the interest is minted to the user through the _mint function, but this interest amount is not added to totalMintedUSIC, resulting in inaccurate total minted amount statistics.

Code Location:

contracts/USICBase.sol#L220-238

```
    function _updateUserGrowthFactor(address user) internal {
        ...

        if (newBalance > currentBalance) {
            uint256 interest = newBalance - currentBalance;
            _mint(user, interest);

            ...
        }

        ...
    }
```

Solution

N/A

Status

Acknowledged; The project team responded that the original design intention is that totalMintedUsIC is only used to record the quantity of USIC tokens minted by users through depositing USDT/USDC, which can be regarded as the principal. Therefore, this situation is in line with expectations and no modification is required.

[N2] [Medium] Potential Hard-Coded Precision Conversion Issues

Category: Arithmetic Accuracy Deviation Vulnerability

Content

In the calculateRedeemableUSDC function of USICBase.sol, this function is used to calculate the amount of USDC that can be obtained after redeeming USIC (after deducting fees). In the mint function of USICCoreUpgradeable.sol, this function is used for users to deposit USDC or USDT to mint USIC tokens at a 1:1 ratio. In the redeem and batchFulfillRedemption functions of USICCoreUpgradeable.sol, these two functions are used to redeem USIC for

USDC. The code repeatedly uses the hard-coded 1e12 for amount conversion. Assuming the precision of USDC/USDT is 6 decimal places, but if deployed on chains like BSC (where the precision of these tokens may be 18 decimal places), it will lead to improper precision conversion and incorrect amount calculations, potentially causing minting/redeeming amounts to exceed normal expectations

Code Location:

contracts/USICBase.sol#L306

```
    function calculateRedeemableUSDC(uint256 amount) public view returns (uint256) {
        if (amount == 0) return 0;

        uint256 fee = calculateRedemptionFee(amount);
        return (amount - fee) / 1e12;
    }
```

contracts/USICCoreUpgradeable.sol#L152&L224&L647

```
    function mint(
        address asset,
        uint256 amount,
        address beneficiary
    ) external nonReentrant whenNotPaused whenAccountNotFrozen(msg.sender)
whenNotSanctioned(msg.sender) updateUserInterest(beneficiary == address(0) ?
msg.sender : beneficiary) {
        ...

        uint256 assetAmount = (amount + 1e12 - 1) / 1e12;

        ...
    }

    function redeem(
        uint256 amount,
        address beneficiary
    ) external nonReentrant whenNotPaused whenRedeemNotPaused
whenAccountNotFrozen(msg.sender) whenNotSanctioned(msg.sender)
updateUserInterest(msg.sender) {
        ...

        uint256 usdcAmount = (amount - fee) / 1e12;

        ...
    }
```

```
    function batchFulfillRedemption(
        address[] calldata users,
        uint256[] calldata amounts
    ) external onlyRole(TREASURY_MANAGER_ROLE) nonReentrant
updateBatchUserInterest(users) {
        ...

        for (uint256 i = 0; i < users.length; i++) {
            ...

            uint256 fee = calculateRedemptionFee(amount);
            uint256 usdcAmount = (amount - fee) / 1e12;

            ...
        }

        ...
    }
```

Solution

It is recommended to replace the hard-coded 1e12 with dynamic calculation. Use IERC20(asset).decimals() to obtain the actual precision of the asset, and calculate the conversion factor such as 10^(18 - decimals) to ensure cross-chain compatibility.

Status

Fixed; Additional note: In the future, the project will be deployed on mainstream layer 2 blockchains like Polygon, Arbitrum and Base. As a result, the precision of USDT/USDC will not exceed 18 decimal places.

[N3] [Low] Lack of Order Verification in Fee Threshold Update

Category: Design Logic Audit

Content

In the _setFeeConfig function of USICCoreAdmin.sol, this function is responsible for updating the fee configuration of a specified index, including the threshold and rate. During the update, feeConfigs[index] is directly assigned, but there is no verification to ensure that the new threshold maintains the increasing order of the array (i.e., greater than the previous configuration threshold and less than the next one, except that the last one can be type(uint256).max). This may disrupt the order of the configuration, affecting the correctness and predictability of the fee calculation

logic. The triggering condition is that the administrator calls _setFeeConfig to set a non-increasing threshold, which impacts all fee-calculation paths that depend on feeConfigs.

Code Location:

src/USICCoreAdmin.sol#L84-95

```
    function _setFeeConfig(
        uint256 index,
        uint256 threshold,
        uint256 feeRate
    ) external {
        ...

        feeConfigs[index] = FeeConfig(threshold, feeRate);

        emit FeeConfigChanged(index, threshold, feeRate, msg.sender);
    }
```

Solution

It is recommended to add threshold order verification logic in _setFeeConfig.

Status

Fixed

[N4] [Suggestion] Redundant Code

Category: Others

Content

1.In the _setAdminModule function of USICCoreAdmin.sol, this function is responsible for setting the address of the administrative module and checking for zero addresses. However, no calls to this function can be found throughout the entire codebase (including USICCoreUpgradeable.sol), which leads to redundant code.

Code Location:

src/USICCoreAdmin.sol#L255-262

```
    function _setAdminModule(address _adminModule) external {
        ...
    }
```

2.In the _withdrawFromYearn function of USICDeFiUpgradeable.sol, this function is responsible for withdrawing a specified amount of USDC from Yearn Finance. In the withdrawal logic, there is a check: if (usdcAmount == 0 || currentShares == 0) return;. However, the non-zero checks for usdcAmount and totalYearnShares have already been handled at the beginning of the function, making this check completely redundant. This may increase code complexity. Although there is no direct security impact, it does affect code quality.

Code Location:

src/USICDeFiUpgradeable.sol#L418

```
    function _withdrawFromYearn(uint256 usdcAmount) internal {
        ...

        if (usdcAmount == 0 || currentShares == 0) return;

        ...
    }
```

Solution

It is recommended to remove the _setAdminModule function and its related event emit to clean up the code. And remove the redundant statement to simplify the _withdrawFromYearn function logic.

Status

Fixed

[N5] [Low] Lack of Automatic Unfreezing in Beneficiary Freeze Check

Category: Design Logic Audit

Content

In the mint and redeem functions of USICCoreUpgradeable.sol, these functions are responsible for minting USIC tokens and redeeming USDC, and they check the freeze status for the beneficiary (recipient). When the beneficiary is not the caller, it directly calls _isAccountFrozen(recipient) to check for freezing, but does not first call the _checkAndAutoUnfreeze function for automatic unfreezing. As a result, if the beneficiary's freeze has expired, it should be automatically unfrozen but is still considered frozen, preventing normal operations. The triggering condition is when the beneficiary's freeze has expired but has not been manually unfrozen, and then mint or redeem is

executed. This affects the path for the beneficiary to receive tokens or USDC, potentially causing legitimate operations to fail.

Code Location:

contracts/USICCoreUpgradeable.sol#L137-140&L208-211

```
    function mint(
        address asset,
        uint256 amount,
        address beneficiary
    ) external nonReentrant whenNotPaused whenAccountNotFrozen(msg.sender)
whenNotSanctioned(msg.sender) updateUserInterest(beneficiary == address(0) ?
msg.sender : beneficiary) {
        ...

        if (recipient != msg.sender) {
            if (_isAccountFrozen(recipient)) revert AccountIsFrozen();
            if (isSanctioned(recipient)) revert AddressSanctioned();
        }

        ...
    }

    function redeem(
        uint256 amount,
        address beneficiary
    ) external nonReentrant whenNotPaused whenRedeemNotPaused
whenAccountNotFrozen(msg.sender) whenNotSanctioned(msg.sender)
updateUserInterest(msg.sender) {
        ...

        if (recipient != msg.sender) {
            if (_isAccountFrozen(recipient)) revert AccountIsFrozen();
            if (isSanctioned(recipient)) revert AddressSanctioned();
        }

        ...
    }
```

Solution

Call _checkAndAutoUnfreeze(recipient) before checking the beneficiary's freeze status to ensure that expired freezes are automatically lifted.

Status

Fixed

[N6] [Suggestion] Lack of Update to the Holder Set

Category: Design Logic Audit

Content

In the redeem, batchFulfillRedemption, transfer and transferFrom functions of USICCoreUpgradeable.sol, these functions are responsible for redeeming USIC, direct transfer and authorized transfer operations. When minting, the beneficiary is added to the _usicHolders set. However, when a user redeems all, or transfers all the balance through transfer/transferFrom resulting in the sender's balance being zero, the corresponding address is not removed from the _usicHolders. This causes the set to contain invalid (zero-balance) holders, affecting the accuracy of holder information tracking.

Code Location:

contracts/USICCoreUpgradeable.sol

```
    function redeem(
        uint256 amount,
        address beneficiary
    ) external nonReentrant whenNotPaused whenRedeemNotPaused
whenAccountNotFrozen(msg.sender) whenNotSanctioned(msg.sender)
updateUserInterest(msg.sender) {
        ...
    }

    function transfer(
        address to,
        uint256 amount
    ) public override whenNotPaused whenAccountNotFrozen(msg.sender)
whenAccountNotFrozen(to) whenNotSanctioned(msg.sender) whenNotSanctioned(to)
updateUserInterest(msg.sender) updateUserInterest(to) returns (bool) {
        ...
    }

    function transferFrom(
        address from,
        address to,
        uint256 amount
    ) public override whenNotPaused whenAccountNotFrozen(from)
whenAccountNotFrozen(to) whenNotSanctioned(from) whenNotSanctioned(to)
```

```
updateUserInterest(from) updateUserInterest(to) returns (bool) {
        ...
    }

   function batchFulfillRedemption(
        address[] calldata users,
        uint256[] calldata amounts
    ) external onlyRole(TREASURY_MANAGER_ROLE) nonReentrant
updateBatchUserInterest(users) {
        ...
    }
```

Solution

It is recommended to check the new balance of the sender (msg.sender or from) after the successful execution of the above functions. If the balance is zero, remove it from the holder set to maintain the accuracy of the set.

Status

Fixed

[N7] [Medium] Lack of Check for Paused Redemption

Category: Design Logic Audit

Content

In the requestRedemption function of USICCoreUpgradeable.sol, this function is responsible for users submitting redemption requests, which are then processed later when the administrator calls the batchFulfillRedemption function. However, neither of these two functions has the whenRedeemNotPaused modifier to check if redemption is paused. As a result, redemption requests can still be submitted when redeemPaused is true, affecting the system's consistency.

Code Location:

contracts/USICCoreUpgradeable.sol#L266-292&L617-702

```
    function requestRedemption(
        uint256 amount
    ) external nonReentrant whenNotPaused whenAccountNotFrozen(msg.sender)
whenNotSanctioned(msg.sender) updateUserInterest(msg.sender) {
        ...
    }

   function batchFulfillRedemption(
```

```
        address[] calldata users,
        uint256[] calldata amounts
    ) external onlyRole(TREASURY_MANAGER_ROLE) nonReentrant
updateBatchUserInterest(users) {
        ...
    }
```

Solution

Add the whenRedeemNotPaused modifier to the requestRedemption and batchFulfillRedemption functions to ensure that new requests are prohibited from being submitted and executed during the paused redemption period.

Status

Fixed

[N8] [Medium] Lack of Verification of User Freeze and Sanction Status in the batchFulfillRedemption Function

Category: Design Logic Audit

Content

In the batchFulfillRedemption function of USICCoreUpgradeable.sol, this function is carried out by the administrator role to batch-process users' redemption requests, including calculating the total USDC demand and preparing the final amount array. The function does not verify the freeze status (whenAccountNotFrozen) and sanction status (whenNotSanctioned) for each address in the users array. This may lead to redeeming for frozen or sanctioned users, thus bypassing the system's compliance controls.

Code Location:

contracts/USICCoreUpgradeable.sol#L617-702

```
   function batchFulfillRedemption(
        address[] calldata users,
        uint256[] calldata amounts
    ) external onlyRole(TREASURY_MANAGER_ROLE) nonReentrant
updateBatchUserInterest(users) {
        ...
    }
```

Solution

It is recommended to check the freeze and sanction status of each user within the loop.

Status

Fixed

[N9] [Suggestion] Optimizable Batch Redemption Amount Check

Category: Gas Optimization Audit

Content

In the batchFulfillRedemption function of USICCoreUpgradeable.sol, this function is responsible for the administrator to batch-process users' redemption requests, including pre - validation and calculation of USDC demand. In the pre-validation loop, each amounts[i] is checked individually to ensure pendingRedemptionRequests[user] >= amount. However, if the same user appears multiple times in the users array, and although the redemption amount each time is less than pendingRedemptionRequests[user], the total redemption amount exceeds the user's pending redemption request, this check will still pass the loop, but a subtraction failure will occur later, resulting in unnecessary gas consumption and inefficient execution.

Code Location:

contracts/USICCoreUpgradeable.sol#L632-653

```
    function batchFulfillRedemption(
        address[] calldata users,
        uint256[] calldata amounts
    ) external onlyRole(TREASURY_MANAGER_ROLE) nonReentrant
updateBatchUserInterest(users) {
        ...

        for (uint256 i = 0; i < users.length; i++) {
            ...

            if (pendingRedemptionRequests[user] < amount) revert
InsufficientPendingRequest();
            if (balanceOf(user) < amount) revert InsufficientBalance();

            ...
        }
```

```
        ...
    }
```

Solution

Before pre-validation, use a mapping to summarize the total redemption amount of each user. Then, check whether the total redemption amount of a single user calculated from the statistics does not exceed pendingRedemptionRequests[user]. Aggregate first before the loop to optimize performance.

Status

Acknowledged; The project team responded: After evaluation, we have decided not to implement this optimization for the following reasons:

Permission Control and Operational Specifications: The batchFulfillRedemption function is called by the TREASURY_MANAGER_ROLE, which is a controlled management operation. We have established operational specifications to ensure that the input array does not contain duplicate users, and we reduce the risk of misoperation through an internal review process.

No Impact on Capital Safety: Even if there are duplicate users, the existing logic will revert due to insufficient balance during the execution phase, and no capital loss will occur. The worst-case scenario is a failed transaction and gas consumption, which is an acceptable operational risk.

Trade-off of Code Complexity: The current implementation is more concise and has a lower maintenance cost. Considering the frequency of use of this function and the actual risks, we prefer to maintain the existing implementation and avoid problems through process control.

Future Optimization Space: If practical problems caused by duplicate users occur in the future, we will re-evaluate and implement an optimization plan.

[N10] [Suggestion] Gas Optimization in the allocateFunds Function

Category: Gas Optimization Audit

Content

In the allocateFunds function of USICDeFiUpgradeable.sol, this function is responsible for allocating USDC to AAVE and Yearn. It transfers aaveAmount and yearnAmount from the core contract to the DeFi contract according to the proportion, resulting in two safeTransferFrom operations, which increases gas consumption. Although there is no direct security hazard, the performance can be optimized.

Code Location:

contracts/USICDeFiUpgradeable.sol#L301-327

```
    function allocateFunds(uint256 usdcAmount) external onlyCore nonReentrant {
        ...
    }
```

Solution

It is recommended to first transfer the total usdcAmount from the core contract to the DeFi contract in one go. Then, calculate the proportion internally and deposit into AAVE and Yearn respectively to reduce the number of transfer operations.

Status

Fixed

[N11] [Medium] Risk of excessive privilege

Category: Authority Control Vulnerability Audit

Content

In the setDeFiContract, emergencyWithdraw, batchMint functions of USICCoreUpgradeable.sol, as well as the setCoreContract, setOTCWallet, setYearnVault, setAavePool, emergencyWithdraw functions of USICDeFiUpgradeable.sol, these functions are responsible for core operations such as setting the addresses of system core contracts, withdrawing funds from the contract, or batch-minting tokens for arbitrary addresses. Since these functions are only controlled by the OPERATOR_ROLE or TREASURY_MANAGER_ROLE, if the private key of the EOA account holding these roles is stolen, it will affect the security and availability of funds in the system.

Code Location:

contracts/USICCoreUpgradeable.sol

```
    function setDeFiContract(address _defiContract) external onlyRole(OPERATOR_ROLE) {
        ...
    }

    function emergencyWithdraw(address token, uint256 amount) external
onlyRole(TREASURY_MANAGER_ROLE) whenPaused nonReentrant {
        ...
    }
```

```
    function batchMint(
         address[] calldata users,
         uint256[] calldata amounts
     ) external onlyRole(OPERATOR_ROLE) nonReentrant whenNotPaused
updateBatchUserInterest(users) {
         ...
     }
```

contracts/USICDeFiUpgradeable.sol

```
    function setCoreContract(address _coreContract) external onlyRole(OPERATOR_ROLE) {
         ...
     }

     function setOTCWallet(address _otcWallet) external onlyRole(DEFAULT_ADMIN_ROLE) {
         ...
     }

     function setYearnVault(address _yearnVault) external onlyRole(OPERATOR_ROLE)
nonReentrant {
         ...
     }

     function setAavePool(address _aavePool) external onlyRole(OPERATOR_ROLE) {
         ...
     }

    function emergencyWithdraw(address token, uint256 amount) external
onlyRole(TREASURY_MANAGER_ROLE) whenPaused nonReentrant {
         ...
     }
```

Solution

In the short term, transferring the ownership of core roles to time-locked contracts and managing them by multi-signatures is an effective solution to avoid single-point risks. However, in the long run, a more reasonable solution is to implement a permission separation strategy and set up multiple privileged roles to manage each privileged function separately. Permissions involving user funds and contract core parameter updates should be managed by the community, while permissions involving emergency contract suspensions can be managed by EOA addresses. This ensures the safety of user funds while responding quickly to threats.

Status

Acknowledged; The project team responded that in the initial stage, we adopt centralized governance. We plan to set the owners of the OPERATOR_ROLE and TREASURY_MANAGER_ROLE as multi-signature wallets (such as Gnosis Safe) to reduce the risk of single-point private key leakage, while retaining the ability to respond quickly to market changes and operational requirements (such as interest rate adjustments, parameter optimizations, and traditional financial asset investment decisions).

In addition, there are the following supplementary security measures: All key operations are recorded with on-chain events for easy monitoring and auditing. An emergency pause function is retained to respond quickly to security threats. Security audits and code reviews are carried out during upgrades, and audits are conducted by multiple vendors. After the product operation stabilizes, we will gradually introduce community governance and time-lock mechanisms. Operations involving user funds and core parameters will be handed over to community governance (such as through a DAO or similar mechanisms), while emergency operations retain the ability for rapid response. We will also create more fine-grained roles (such as CONFIG_ROLE, TREASURY_ROLE, EMERGENCY_ROLE) to be managed by different governance mechanisms. We will maintain operational transparency, regularly release operation reports and fund audit reports. For the reserves of national bonds and high-rated corporate bonds, we will cooperate with accounting firms to issue reserve audit reports regularly. We will maintain communication with the community and disclose important changes in a timely manner to ensure that users fully understand the project architecture and security measures.

Updated: The core contract role's permissions have been transferred to the following multi-signature wallets:

https://etherscan.io/address/0xf8B425a062F8249767957F1E9798AEd063D6FE72

https://etherscan.io/address/0xAd5466582428ea71655b0815F09A833e73702d2B

The transfer tx links are as follows:

https://etherscan.io/tx/0xd26946065ae085deafabeafc1b7d4c5c4742ee65ee884a5cd5cdff5ee208bd6c

https://etherscan.io/tx/0xe6d92793734354f162f8424ea7acbfa6f95389ee509e90b057124d1d5ecc88bb

https://etherscan.io/tx/0x039f85a80d1da36e1bde22def1bdf68b3089b52f7016f2aabb715a58fe336c73

https://etherscan.io/tx/0x9caba26fc0eb62f7e0d313f56918f96a0e67b7b49c30a8cd1f3dfb879c391f74

https://etherscan.io/tx/0x3eb604ccc36f991aaec70d82f967560e8696aaa268223ef241f849f6327de30a

[N12] [Information] Potential Bank Run Situation

Category: Others

Content

When users deposit USDC or USDT to mint USIC tokens, the deposited funds are invested externally to generate returns. 80% of the funds are temporarily stored in Aave and then transferred to traditional finance for purchasing national bonds, and this part of the funds is managed by the project team's OTC wallet. The other 20% of the funds are deposited in the pool of Yearn Finance for investment and are directly managed by the contract.

When the redeem function is called to directly redeem assets, if the existing USDC balance in the contract is insufficient, withdrawals will be made from Yearn Finance. This may lead to a situation where, in case of large-scale batch redemptions, if the USDC deposited in Yearn Finance is not enough to meet the redemption amount, some users may not be able to redeem normally, that is, a bank run situation.

The current mechanism of the project team for this situation is that users will see the total amount of funds that can be redeemed immediately in the front-end (the funds in Yearn Finance). If it is less than the amount they want to redeem, the front-end page will prompt them that "direct redemption is unavailable because the current liquid funds are insufficient. You need to submit a redemption request and wait for 7 working days. You will automatically receive the redeemed USDC after 7 days." Finally, the project team will liquidate the national bonds offline and directly transfer the redeemed USDC to the users.

Solution

N/A

Status

Acknowledged

[N13] [Information] Potential External Protocol Risks

Category: Others

Content

In the allocateFunds function of USICDeFiUpgradeable.sol, this function is responsible for allocating the USDC deposited by users to the Aave and Yearn Finance protocols for investment to generate returns. However, if the external protocols are attacked by vulnerabilities or face other risks, it will lead to losses of users' funds, which may in turn cause the USIC stablecoin to de-peg. Furthermore, most of the funds deposited by users will be managed by

the project team's OTC wallet. Caution is needed regarding the potential centralized risks that the OTC wallet may encounter, so as to avoid loss of funds.

Solution

N/A

Status

Acknowledged

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002511170001	SlowMist Security Team	2025.11.10 - 2025.11.17	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 4 medium, 2 low risks, 4 suggestion and 3 information. All the findings were fixed or acknowledged.

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