Smart Contract Security Audit Report

Table Of Contents

1 Executive Summary

On 2024.11.28, the SlowMist security team received the USDX Money team's security audit application for USDX Contracts, developed the audit plan according to the agreement of both parties and the characteristics of the project, and finally issued the security audit report.

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

The test method information:

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

The vulnerability severity level information:

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

2 Audit Methodology

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

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

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

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

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

3 Project Overview

3.1 Project Introduction

USDX is a stablecoin eco-protocol, a project that implements its core functionality through a series of smart contracts, including minting, selling, redeeming, staking, and distributing rewards to liquidity providers.

Key Components:

`USDX.sol` : As the core contract for Stablecoin, it defines the basic rules of Stablecoin and introduces a Minter role, which is authorized by the contract Owner. This role allows an address-specific (e.g. USDXSales contract) to perform asset exchange operations, such as converting assets like USDC to USDX.

`USDXSales.sol` : Acts as a minting role for USDX and allows users to exchange other supported stablecoins for newly minted USDX through this contract.

`USDXRedeem.sol` : Users can use this redemption contract to exchange USDX for supported assets. This increases the flexibility and utility of USDX in the market.

`StakedUSDX.sol` : This is where USDX holders stake their stablecoin to receive proceeds in the form of sUSDX. The distribution of proceeds in the protocol is handled by the staking contract with the role of REWARDER, which ensures fairness through a linear release mechanism and prevents preemptive trading behavior. In addition, in order to comply with legal requirements, the contract incorporates different levels of restrictions, imposing specific access controls on participants in certain regions. Meanwhile, when a user chooses to release a stake, there is a cooling-off period ranging from 7 to 90 days, during which the funds are temporarily kept in a separate custodial contract.

`USDXLPStaking.sol` : A staking contract designed for liquidity providers that allows them to earn Ethena airdrop shares by staking LP tokens in different USDX liquidity pools. A specific pool of liquidity is eligible for each period, while the calculation and distribution of rewards is handled off-chain.

These components of the USDX project work together to form a complete stablecoin ecosystem that not only provides traditional stablecoin functionality, but also combines DeFi features such as staking interest generation and liquidity mining. It is worth noting that while USDX itself is not subject to any freezing or restriction policies, its staking contracts contain some centralized management elements, such as the ability to freeze and recapture funds from restricted addresses, for compliance reasons. As a result, users need to have some basis of trust in the Synth-X organization when participating in USDX-related activities.

3.2 Vulnerability Information

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

NO	Title	Category	Level	Status
N1	Potential risk of funds being locked	Design Logic Audit	High	Acknowledged
N2	Missing key role checks	Authority Control Vulnerability Audit	Medium	Acknowledged
N3	Improper exchange rate calculation method	Arithmetic Accuracy Deviation Vulnerability	Medium	Acknowledged

NO	Title	Category	Level	Status
N4	Potential risk of token compatibility	Design Logic Audit	Suggestion	Acknowledged
N5	Potential DOS risk via permit front-running	Denial of Service Vulnerability	Low	Acknowledged
N6	Missing non-zero address check	Others	Suggestion	Acknowledged
N7	Risk of excessive authority	Authority Control Vulnerability Audit	Medium	Acknowledged

4 Code Overview

4.1 Contracts Description

Audit Version:

https://github.com/Synth-X/usdx-contract/tree/main

commit: 15f446f6fbc366ec4e134cca9874eeb2b7566548

Audit scope:

```
./contracts
├── Migrations.sol
├── StakedUSDX.sol
├── USDX.sol
├── USDXLPStaking.sol
├── USDXRedeem.sol
├── USDXSales.sol
├── USDXSilo.sol
├── helpers
├── misc
└── interfaces
```

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:

Migrations			
Function Name	Visibility	Mutability	Modifiers
setCompleted	Public	Can Modify State	restricted

StakedUSDX			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	ERC20 ERC4626 ERC20Permit
transferInRewards	External	Can Modify State	nonReentrant onlyRole notZero
deposit	Public	Can Modify State	whenNotDepositPaused
depositWithPermit	Public	Can Modify State	whenNotDepositPaused
mint	Public	Can Modify State	whenNotDepositPaused
mintWithPermit	Public	Can Modify State	whenNotDepositPaused
withdraw	Public	Can Modify State	whenNotWithdrawPaused ensureCooldownOff
redeem	Public	Can Modify State	whenNotWithdrawPaused ensureCooldownOff
unstake	External	Can Modify State	whenNotWithdrawPaused
cooldownAssets	External	Can Modify State	whenNotWithdrawPaused ensureCooldownOn
cooldownShares	External	Can Modify State	whenNotWithdrawPaused ensureCooldownOn
setCooldownDuration	External	Can Modify State	onlyOwner

StakedUSDX			
configPoolLimit	External	Can Modify State	onlyOwner
pauseDeposit	External	Can Modify State	onlyOwner
unpauseDeposit	External	Can Modify State	onlyOwner
pauseWithdraw	External	Can Modify State	onlyOwner
unpauseWithdraw	External	Can Modify State	onlyOwner
isBlacklist	Public	-	-
addToBlacklist	External	Can Modify State	onlyRole notOwner
removeFromBlacklist	External	Can Modify State	onlyRole notOwner
isRewarder	External	-	-
addRewarder	External	Can Modify State	onlyOwner
removeRewarder	External	Can Modify State	onlyOwner
isBlacklistManager	External	-	-
addBlacklistManager	External	Can Modify State	onlyOwner
removeBlacklistManager	External	Can Modify State	onlyOwner
rescueTokens	External	Can Modify State	onlyOwner
redistributeLockedAmount	External	Can Modify State	onlyOwner
totalAssets	Public	-	-
getUnvestedAmount	Public	-	-
decimals	Public	-	-
_checkMinShares	Internal	-	-

StakedUSDX			
_deposit	Internal	Can Modify State	nonReentrant notZero notZero
_withdraw	Internal	Can Modify State	nonReentrant notZero notZero
_updateVestingAmount	Internal	Can Modify State	-
_beforeTokenTransfer	Internal	Can Modify State	-
renounceRole	Public	Can Modify State	-

USDX			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	ERC20 ERC20Permit
mint	External	Can Modify State	onlyRole
isMinter	Public	-	-
addMinter	External	Can Modify State	onlyOwner
removeMinter	External	Can Modify State	onlyOwner
renounceOwnership	Public	-	onlyOwner

USDXLPStaking			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
setEpoch	External	Can Modify State	onlyOwner
updateStakeParameters	External	Can Modify State	onlyOwner
rescueTokens	External	Can Modify State	onlyOwner nonReentrant checkAmount
renounceOwnership	Public	-	onlyOwner

USDXLPStaking			
stake	External	Can Modify State	nonReentrant checkAmount
unstake	External	Can Modify State	nonReentrant checkAmount
withdraw	External	Can Modify State	nonReentrant checkAmount
_checkInvariant	Internal	-	-

USDXRedeem			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
redeem	External	Can Modify State	nonReentrant whenNotPaused
redeemWithPermit	External	Can Modify State	nonReentrant whenNotPaused
claim	External	Can Modify State	nonReentrant whenNotPaused
setCooldownDuration	External	Can Modify State	onlyOwner
updateFeeRate	External	Can Modify State	onlyOwner
updateMaxPerRedeem	External	Can Modify State	onlyOwner
updateVault	External	Can Modify State	onlyOwner
pause	External	Can Modify State	onlyOwner
unpause	External	Can Modify State	onlyOwner
addSupportedAsset	Public	Can Modify State	onlyOwner
removeSupportedAsset	External	Can Modify State	onlyOwner
isSupportedAsset	External	-	-
listSupportedAssets	Public	-	-
rescueTokens	External	Can Modify State	onlyOwner
_redeem	Internal	Can Modify State	-

USDXRedeem			
_payOrTransfer	Internal	Can Modify State	-

USDXSales			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
buy	External	Can Modify State	whenNotPaused nonReentrant
buyWithPermit	External	Can Modify State	whenNotPaused nonReentrant
addSupportedAsset	Public	Can Modify State	onlyOwner
addCustodianAddress	Public	Can Modify State	onlyOwner
removeSupportedAsset	External	Can Modify State	onlyOwner
removeCustodianAddress	External	Can Modify State	onlyOwner
isSupportedAsset	External	-	-
isCustodianAddress	External	-	-
listSupportedAssets	Public	-	-
listCustodians	Public	-	-
updateFeeRate	External	Can Modify State	onlyOwner
setMaxMintPerBlock	External	Can Modify State	onlyOwner
pause	External	Can Modify State	onlyOwner
unpause	External	Can Modify State	onlyOwner
_buy	Internal	Can Modify State	-
_setMaxMintPerBlock	Internal	Can Modify State	-

USDXSilo			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
withdraw	External	Can Modify State	onlyStakingVault

Timelock			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
<Receive Ether>	External	Payable	-
<Fallback>	External	Payable	-
setDelay	Public	Can Modify State	-
acceptAdmin	Public	Can Modify State	-
setPendingAdmin	Public	Can Modify State	-
queueTransaction	Public	Can Modify State	-
cancelTransaction	Public	Can Modify State	-
executeTransaction	Public	Payable	-
getBlockTimestamp	Public	-	-
getTxHash	Public	-	-

4.3 Vulnerability Summary

[N1] [High] Potential risk of funds being locked

Category: Design Logic Audit

Content

In the StakedUSDX contract, the _checkMinShares function is used to check that very small amounts of shares will not be left in the contract to prevent interest rate inflation attacks. However, this function is also called for checking in

the _withdraw function, which might lead to a situation where if a user is the last to withdraw, and the previous user deliberately left a very small amount of shares in the contract, it would result in the user being unable to withdraw normally.

For example:

Suppose the contract already has two users who have deposited funds, namely Alice and Bob, each depositing an amount of 1e18 USDX tokens, making the totalSupply at this point 2e18.

1.Assume Alice now wants to call the withdraw function to initiate a withdrawal, extracting an amount of 1e18 USDX tokens.

2.At this moment, Bob gets ahead and calls the withdraw function to initiate a withdrawal, extracting an amount of (1e18 - 1) tokens. This can successfully pass the check of the _checkMinShares function because after Bob's withdrawal, the totalSupply becomes 1e18 + 1.

3.And then when it's Alice's turn for her withdrawal transaction to be executed, since the totalSupply in the contract becomes 1 after the withdrawal, it will not pass the _checkMinShares function check. This will result in Alice never being able to withdraw her funds from the contract, unless new funds are deposited into the contract.

Code Location:

contracts/StakedUSDX.sol#L421

```
    function _checkMinShares() internal view {
        uint256 _totalSupply = totalSupply();
        if (_totalSupply > 0 && _totalSupply < MIN_SHARES)
revert(Errors.MIN_SHARES_VIOLATION);
    }

    ...

    function _withdraw(
      address caller,
      address receiver,
      address _owner,
      uint256 assets,
      uint256 shares
    ) internal override nonReentrant notZero(assets) notZero(shares) {
      ...

      _checkMinShares();
```

```
    }
```

Solution

It is recommended to upgrade the used OpenZeppelin library to the latest version and use the latest version of ERC4626 to avoid the interest rate inflation vulnerability, instead of using the _checkMinShares function for checking.

Status

Acknowledged;

Project team response: Expected design. To prevent the StakedUSDX pool from draining to zero, the project team will ensure a minimum reserve is always kept in the pool, so users can always withdraw their funds.

[N2] [Medium] Missing key role checks

Category: Authority Control Vulnerability Audit

Content

According to the design document provided by the project team, there are two special roles in the StakedUSDX contract: SOFT_RESTRICTED_STAKER_ROLE and FULL_RESTRICTED_STAKER_ROLE.

The former is for addresses based in countries we are not allowed to provide yield to, for example USA. Addresses under this category will be soft restricted. They cannot deposit USDX to get sUSDX or withdraw sUSDX for USDX. However, they can participate in earning yield by buying and selling sUSDX on the open market.

However, in the contract's redistributeLockedAmount and _withdraw functions, there is no check to see if the provided address has the SOFT_RESTRICTED_STAKER_ROLE. This means that addresses with the SOFT_RESTRICTED_STAKER_ROLE can also earn profits through channels other than buying and selling, which does not align with the expected design outlined in the documentation.

Code Location:

contracts/StakedUSDX.sol

```
    function redistributeLockedAmount(address from, address to) external onlyOwner {
        ...
    }

    ...
```

```
    function _withdraw(
      address caller,
      address receiver,
      address _owner,
      uint256 assets,
      uint256 shares
    ) internal override nonReentrant notZero(assets) notZero(shares) {
        ...
    }
```

Solution

It is recommended to check if the provided address parameter has the SOFT_RESTRICTED_STAKER_ROLE role in the redistributeLockedAmount and _withdraw functions.

Status

Acknowledged; Project team response: Expected designed. The redistributeLockedAmount and withdraw functions will restrict the FULL_RESTRICTED_STAKER_ROLE. And the description in the readme document has been modified.

[N3] [Medium] Improper exchange rate calculation method

Category: Arithmetic Accuracy Deviation Vulnerability

Content

In the USDXSales contract, When minting USDX by calling the _buy function, the amount of USDX minted is calculated by multiplying the amount of collateral tokens after deducting fees by 1e18 and then dividing it by the precision of the collateral token. If the precision of the collateral token is greater than 1e18, this will result in the calculated amount of USDX tokens being 0, while the collateral has already been transferred to the _custodianAddress, causing a loss of user assets. The same issue also exists in the USDXRedeem contract.

Code Location:

contracts/USDXSales.sol#L150

```
  function _buy(address _collateralAsset, uint256 _collateralAmount, address
_custodianAddress) internal {
    ...

    IERC20(_collateralAsset).safeTransferFrom(_msgSender(), _custodianAddress,
_collateralAmount);

    uint256 _usdxAmount = (_collateralAmount - fee) * (1e18 / 10 **
```

```
IERC20Metadata(_collateralAsset).decimals());
    ...
  }
```

contracts/USDXRedeem.sol#L170

```
  function _redeem(address _assetToken, uint256 _usdxAmount) internal {
    uint256 _assetAmount = _usdxAmount / (1e18 / 10 **
IERC20Metadata(_assetToken).decimals());

    ...

    IERC20(address(USDX)).safeTransferFrom(_msgSender(), address(this), _usdxAmount);

    emit Redeem(_msgSender(), _assetToken, _assetAmount, _usdxAmount, fee);
  }
```

Solution

It is recommended to check that the calculated result is not equal to 0, and during the calculation, you should adopt the method of multiplying by 1e18 first and then dividing by the precision, instead of directly multiplying by the result of the division.

Status

Acknowledged; Project team response: All assets supported by USDXSales and USDXRedeem require project team approval; therefore, this special precision scenario does not exist.

[N4] [Suggestion] Potential risk of token compatibility

Category: Design Logic Audit

Content

In the USDXSales contract, When the _buy function is called to mint USDX tokens, the collateral token will be first transferred into the contract. Then, it directly uses the passed collateral amount parameter _collateralAmount to participate in the calculation of the amount of USDX tokens to be minted. However, the function does not check whether the difference in the contract's token balance before and after the transfer equals the value of the _collateralAmount parameter. If the token is a deflationary token, it might result in the actual amount transferred being less than the value of amount, ultimately leading to unexpected errors.

Code Location:

contracts/USDXSales.sol#L148

```
  function _buy(address _collateralAsset, uint256 _collateralAmount, address
_custodianAddress) internal {
    ...

    IERC20(_collateralAsset).safeTransferFrom(_msgSender(), _custodianAddress,
_collateralAmount);

    uint256 _usdxAmount = (_collateralAmount - fee) * (1e18 / 10 **
IERC20Metadata(_collateralAsset).decimals());
    ...
  }
```

Solution

It is recommended to use the difference in the token balance in the contract before and after the user's transfer as the actual _collateralAmount, instead of directly using the _collateralAmount parameter passed in.

Status

Acknowledged; Project team response: All supported assets require approval from the project team; therefore, inflationary tokens do not exist in this context.

[N5] [Low] Potential DOS risk via permit front-running

Category: Denial of Service Vulnerability

Content

Once the permit data is submitted, it can be publicly accessed in the memory pool, and anyone can execute the permit by copying the transaction parameters. After calling permit(), a second call with the same parameters will revert.

Thus, a malicious attacker can obtain the parameters passed by other users for calling permit-related functions (such as buyWithPermit) in the memory pool, front-run to activate this permit, bypass the contract's buyWithPermit function, and ultimately cause the normal user's buyWithPermit() transaction to fail.

Reference:

https://www.trust-security.xyz/post/permission-denied

Code Location:

contracts/USDXSales.sol#L79

```
  function buyWithPermit(
    address _collateralAsset,
    uint256 _collateralAmount,
    address _custodianAddress,
    uint256 _deadline,
    uint8 _permitV,
    bytes32 _permitR,
    bytes32 _permitS
  ) external override whenNotPaused nonReentrant {
    ...

    IERC20Permit(_collateralAsset).safePermit(_msgSender(), address(this),
_collateralAmount, _deadline, _permitV, _permitR, _permitS);

    ...
  }
```

contracts/USDXRedeem.sol#L80

```
  function redeemWithPermit(
    address _assetToken,
    uint256 _usdxAmount,
    uint256 _deadline,
    uint8 _permitV,
    bytes32 _permitR,
    bytes32 _permitS
  ) external override nonReentrant whenNotPaused {
    ...

    IERC20Permit(address(USDX)).safePermit(_msgSender(), address(this), _usdxAmount,
_deadline, _permitV, _permitR, _permitS);

    ...
  }
```

contracts/StakedUSDX.sol#L141&L166

```
  function depositWithPermit(
    uint256 assets,
    address receiver,
    uint256 _deadline,
```

```
    uint8 _permitV,
    bytes32 _permitR,
    bytes32 _permitS
  ) public whenNotDepositPaused returns (uint256) {
    IERC20Permit(asset()).safePermit(_msgSender(), address(this), assets, _deadline,
_permitV, _permitR, _permitS);

    ...
  }

  ...

  function mintWithPermit(
    uint256 shares,
    address receiver,
    uint256 _deadline,
    uint8 _permitV,
    bytes32 _permitR,
    bytes32 _permitS
  ) public whenNotDepositPaused returns (uint256) {
    ...

    IERC20Permit(asset()).safePermit(_msgSender(), address(this), assets, _deadline,
_permitV, _permitR, _permitS);

    ...
  }
```

Solution

It is recommended to using the try/catch pattern for permit operations to prevent reverts.

Status

Acknowledged

[N6] [Suggestion] Missing non-zero address check

Category: Others

Content

The admin role can set critical address variables in several functions, but there is a lack of non-zero address validation checks for the parameters passed in.

Code Location:

contracts/StakedUSDX.sol#L281&L326

```
  function addToBlacklist(address target, bool isFullBlacklisting) external override
onlyRole(BLACKLIST_MANAGER_ROLE) notOwner(target) {
    bytes32 role = isFullBlacklisting ? FULL_RESTRICTED_STAKER_ROLE :
SOFT_RESTRICTED_STAKER_ROLE;
    _grantRole(role, target);
  }

  function rescueTokens(address token, uint256 amount, address to) external onlyOwner
{
    require(address(token) != asset(), Errors.INVALID_TOKEN);
    IERC20(token).safeTransfer(to, amount);
  }
```

contracts/USDXSilo.sol#L29

```
  function withdraw(address to, uint256 amount) external onlyStakingVault {
    USDX.transfer(to, amount);
  }
```

Solution

It is recommended that the address non-zero checks should be added.

Status

Acknowledged

[N7] [Medium] Risk of excessive authority

Category: Authority Control Vulnerability Audit

Content

1.In the StakedUSDX contract, the owner role can set the cooldown period and pool maximum limits regarding staking operations by calling the setCooldownDuration and configPoolLimit functions. Additionally, the owner can invoke the rescueTokens function to transfer all tokens within the contract except USDX out, and can call the redistributeLockedAmount function to migrate any person's LP tokens. The blacklist manager role can add or remove any address from the blacklist. If the privileges are lost or misused, this may have an impact on the user’s assets.

Code Location:

contracts/StakedUSDX.sol

```
  function setCooldownDuration(uint24 duration) external override onlyOwner {
    ...
  }

  function configPoolLimit(uint256 _poolDepositLimit) external override onlyOwner {
    ...
  }

  function addToBlacklist(address target, bool isFullBlacklisting) external override
onlyRole(BLACKLIST_MANAGER_ROLE) notOwner(target) {
    ...
  }

  function removeFromBlacklist(
    address target,
    bool isFullBlacklisting
  ) external override onlyRole(BLACKLIST_MANAGER_ROLE) notOwner(target) {
    ...
  }

  function rescueTokens(address token, uint256 amount, address to) external onlyOwner
{
    ...
  }

  function redistributeLockedAmount(address from, address to) external onlyOwner {
    ...
  }
```

2.In the USDX contract, the minter role can mint USDX tokens to any address by calling the mint function. If the privileges are lost or misused, it could lead to a large amount of USDX being minted and flooding the market, affecting regular users.

Code Location:

contracts/USDX.sol

```
  function mint(address to, uint256 amount) external onlyRole(MINTER_ROLE) {
    _mint(to, amount);
  }
```

3.In the USDXLPStaking contract, the owner role can call the updateStakeParameters function to set the configuration items for token staking, including the staking epoch, limit, and cooldown period. Additionally, the owner

can call the rescueTokens function to transfer tokens from the contract to any address. If the privileges are lost or misused, this may have an impact on the user's assets.

Code Location:

contracts/USDXLPStaking.sol

```
  function updateStakeParameters(address token, uint8 epoch, uint248 stakeLimit,
uint48 cooldown) external onlyOwner {
    ...
  }

  function rescueTokens(address token, address to, uint256 amount) external onlyOwner
nonReentrant checkAmount(amount) {
    ...
  }
```

4.In the USDXRedeem contract, the owner role can call the setCooldownDuration function and the updateMaxPerRedeem function to set the cooldown period for redemption and the maximum limit per redemption, respectively. Additionally, the owner can call the rescueTokens function to transfer tokens from the contract to any address, including the collateral tokens.

Code Location:

contracts/USDXRedeem.sol

```
  function setCooldownDuration(uint24 _cooldownDuration) external override onlyOwner
{
    ...
  }

  function updateMaxPerRedeem(uint256 _maxRedeem) external override onlyOwner {
    ...
  }

  function rescueTokens(address _token, address _to, uint256 _amount) external
onlyOwner {
    ...
  }
```

Solution

In the short term, during the early stages of the project, the protocol may need to frequently set various parameters

to ensure the stable operation of the protocol. Therefore, transferring the ownership of core roles to a multisig management can effectively solve the single-point risk, but it cannot mitigate the excessive privilege risk. In the long run, after the protocol stabilizes, transferring the owner ownership to community governance or executing through a timelock can effectively mitigate the excessive privilege risk and increase the community users' trust in the protocol.

Status

Acknowledged; Project team response: In the future, Timelock contracts and other methods will be used for permission management.

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002412030001	SlowMist Security Team	2024.11.28 - 2024.12.03	Medium Risk

Summary conclusion: The SlowMist security team use a manual and SlowMist team's analysis tool to audit the project, during the audit work we found 1 high risk, 3 medium risk, 1 low risk and 2 suggestion. All the findings were acknowledged. The code was not deployed to the mainnet. The current risk level is temporarily rated as medium because the centralized control permissions have not yet been managed.

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