Smart Contract Security Audit Report

Table Of Contents

1 Executive Summary

On 2025.07.01, the SlowMist security team received the UXUY Protocol team's security audit application for Uxuy Smart Pool, developed the audit plan according to the agreement of both parties and the characteristics of the project, and finally issued the security audit report.

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

The test method information:

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

The vulnerability severity level information:

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

2 Audit Methodology

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

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

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

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

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

3 Project Overview

3.1 Project Introduction

This audit mainly focuses on the UxuySmartPool contract, which is a cross-chain atomic swap liquidity pool. It aims to enable users to lock assets on one chain and receive equivalent assets on another chain, thus achieving cross-chain transfer and exchange of assets. The project relies on a centralized administrator (Admin) role to verify and execute cross-chain operations, ensuring the final completion of transactions or secure refunds.

3.2 Vulnerability Information

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

NO	Title	Category	Level	Status
N1	Token compatibility reminder	Design Logic Audit	Suggestion	Acknowledged
N2	Potential residual approval amount	Design Logic Audit	Medium	Acknowledged
N3	Lack of checks on external call data	Design Logic Audit	Suggestion	Acknowledged
N4	Missing Slippage Protection	Design Logic Audit	Medium	Acknowledged
N5	Risk of excessive authority	Authority Control Vulnerability Audit	Medium	Acknowledged
N6	Expired failed orders cannot be refunded	Design Logic Audit	Suggestion	Acknowledged

4 Code Overview

4.1 Contracts Description

Audit Version:

https://github.com/uxuycom/smart_wallet_contracts/blob/main/ethereum/contracts/UxuySmartPool.sol

commit: 3ab31b302cf2673b9e8f87a149a99843f1af6a5e

The main network address of the contract is as follows:

https://basescan.org/address/0xCa4fe225fD2c7C81c7196D0Aa081f9003CA083DF

https://bscscan.com/address/0xCa4fe225fD2c7C81c7196D0Aa081f9003CA083DF

https://etherscan.io/address/0xCa4fe225fD2c7C81c7196D0Aa081f9003CA083DF

4.2 Visibility Description

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

UxuySmartPool			
Function Name	Visibility	Mutability	Modifiers

UxuySmartPool			
<Constructor>	Public	Can Modify State	Ownable
pause	External	Can Modify State	onlyOwner
unpause	External	Can Modify State	onlyOwner
addAdmin	External	Can Modify State	onlyOwner
addAdmins	External	Can Modify State	onlyOwner
removeAdmin	External	Can Modify State	onlyOwner
setOperator	External	Can Modify State	onlyOwner
rebalanceOut	External	Can Modify State	whenNotPaused nonReentrant onlyOperator
depositLiquidity	External	Can Modify State	whenNotPaused nonReentrant
withdrawLiquidity	External	Can Modify State	whenNotPaused nonReentrant
swapIn	External	Can Modify State	whenNotPaused nonReentrant
swapOut	External	Can Modify State	whenNotPaused nonReentrant onlyAdmin
swapConfirm	External	Can Modify State	whenNotPaused nonReentrant onlyAdmin

4.3 Vulnerability Summary

[N1] [Suggestion] Token compatibility reminder

Category: Design Logic Audit

Content

In the UxuySmartPool contract, The safeTransferFrom function is used to transfer in the specified ERC20 tokens when calling the depositLiquidity and swapIn functions. However, this function does not check the difference between the balance before and after the transfer to the recipient address and the actual transferred amount. If the token to be transferred is a deflationary token, the actual number of tokens received will not match the number of tokens recorded in the contract.

Code Location:

ethereum/contracts/UxuySmartPool.sol#L112-117&L128-158

```
    function depositLiquidity(address token, uint256 amount) external whenNotPaused
nonReentrant {
        require(amount > 0, "Invalid amount");
        IERC20(token).safeTransferFrom(msg.sender, address(this), amount);
        userBalances[msg.sender][token] += amount;
        emit LiquidityDeposited(msg.sender, token, amount);
    }

    function swapIn(
        address token,
        uint256 amount,
        uint256 targetChainId,
        address targetReceiver,
        bytes32 hashlock,
        uint256 timelock
    ) external whenNotPaused nonReentrant{
        ...

        IERC20(token).safeTransferFrom(msg.sender, address(this), amount);

        orders[hashlock] = SwapOrder({
            token: token,
            sender: msg.sender,
            amount: amount,
            targetChainId: targetChainId,
            targetReceiver: targetReceiver,
            hashlock: hashlock,
            timelock: timelock,
            confirmed: false,
            refunded: false
        });

        emit SwapIn(msg.sender, token, amount, targetChainId, targetReceiver,
hashlock, timelock);
    }
```

Solution

It is recommended to use the difference between the recipient address's balance before and after the transfer to record the user's actual transfer amount.

Status

Acknowledged; The project team responded: This is a pool contract only for stablecoins. It will only support specific currency types and will not support deflationary tokens.

[N2] [Medium] Potential residual approval amount

Category: Design Logic Audit

Content

In the UxuySmartPool contract, the swapOut function is used to directly transfer tokens in the contract to the receiving address or swapping them in an external DEX protocol. When the swapTarget is not equal to the zero address, the tokens will first be authorized to the spender address, and then the swapTarget contract will be externally called for the swap operation according to the passed-in swapCallData. However, the swapOut function does not verify the parameters in the swapCallData. If the amount of tokens actually required for the transaction in the swapCallData is less than the amount parameter, after swapping, it may result in the contract having the remaining approval amount for the spender address. This will pose a risk of these tokens being stolen.

Code Location:

ethereum/contracts/UxuySmartPool.sol#L177-191

```
function swapOut(
     address token,
     uint256 amount,
     address swapTarget,
     address spender,
     bytes calldata swapCallData,
     address receiver,
     bytes32 hashlock,
     bytes calldata preimage
 ) external whenNotPaused nonReentrant onlyAdmin {
     ...
     } else {
         require(swapTarget.code.length > 0, "Invalid swap address");
         // Record original token balance
         uint256 beforeBalance = IERC20(token).balanceOf(address(this));
         IERC20(token).forceApprove(spender, amount);
         (bool success,) = swapTarget.call(swapCallData);
         require(success, "Swap failed");

         // Check how much token was actually used
         uint256 afterBalance = IERC20(token).balanceOf(address(this));
         uint256 usedAmount = beforeBalance - afterBalance;
```

```
            require(usedAmount <= amount, "Swap exceeded limit");

            emit SwapOut(token, usedAmount, receiver, hashlock, preimage);
        }
    }
```

Solution

It is recommended that after the token swap is completed, the approval amount of the spender address should be cleared to zero, and that the quantity of tokens to be traded parsed from the swapCallData parameter be checked to ensure it meets expectations.

Status

Acknowledged; The project team responded that since swapOut can only be called by the owner and the target address is also a platform-trusted address, there is no risk for the time being. They will address this issue in subsequent contract upgrades.

[N3] [Suggestion] Lack of checks on external call data

Category: Design Logic Audit

Content

In the UxuySmartPool contract, when the swapOut function is called for token swap operations, the passed-in swapTarget address will be called, and the function to be called and specific data are all in the externally passed-in swapCallData. However, both the swapTarget parameter and the function signature to be called in the swapCallData parameter lack whitelist checks. If the admin, due to mistakes or other reasons, passes in a swapTarget parameter and a function signature in the swapCallData parameter that do not meet expectations, it may cause losses to the funds deposited by users.

Code Location:

ethereum/contracts/UxuySmartPool.sol#L182

```
    function swapOut(
        address token,
        uint256 amount,
        address swapTarget,
        address spender,
        bytes calldata swapCallData,
        address receiver,
```

```
        bytes32 hashlock,
        bytes calldata preimage
    ) external whenNotPaused nonReentrant onlyAdmin {
        ...

        if(swapTarget == address(0)){
            ...
        } else {
            ...

            (bool success,) = swapTarget.call(swapCallData);

            ...
        }
    }
```

Solution

It is recommended to add whitelist checks for the function signatures to be called in the swapCallData parameter and for the swapTarget parameter.

Status

Acknowledged; The project team responded that the contract's liquidity is basically provided by the platform itself to assist users in cross-chain transactions. Therefore, whitelist control cannot be implemented, and off-chain judgment can only be carried out by the owner role.

[N4] [Medium] Missing Slippage Protection

Category: Design Logic Audit

Content

In the UxuySmartPool contract, when the swapOut function is called for external token swapping, the target tokens will be directly given to the user. However, there is a lack of slippage check on the amount of tokens received by the user at this point, which may cause the user's tokens to be lost due to a sandwich attack.

ethereum/contracts/UxuySmartPool.sol#L177-191

```
    function swapOut(
        address token,
        uint256 amount,
        address swapTarget,
        address spender,
        bytes calldata swapCallData,
```

```
        address receiver,
        bytes32 hashlock,
        bytes calldata preimage
    ) external whenNotPaused nonReentrant onlyAdmin {
        ...
        } else {
            require(swapTarget.code.length > 0, "Invalid swap address");
            // Record original token balance
            uint256 beforeBalance = IERC20(token).balanceOf(address(this));
            IERC20(token).forceApprove(spender, amount);
            (bool success,) = swapTarget.call(swapCallData);
            require(success, "Swap failed");

            ...
        }
    }
```

Solution

It is recommended to add a slippage check on the amount of tokens that users can ultimately receive.

Status

Acknowledged; The project team responded that the slippage check has been set in swapdata. The pool contract itself doesn't care about the rationality of swap parameters, which is handled off-chain by the admin.

[N5] [Medium] Risk of excessive authority

Category: Authority Control Vulnerability Audit

Content

In the UxuySmartPool contract, the owner role can set the admin and the operator roles by calling the addAdmin, addAdmins and setOperator functions. Among them, the Operator role can directly transfer out the tokens in the contract by calling the rebalanceOut function. The Admin role can directly transfer the tokens in the contract to a specified address or exchange them in an external contract by calling the swapOut function. If these roles are set to the EOA addresses and their permissions are compromised, it will cause the loss of users' funds.

Code Location:

ethereum/contracts/UxuySmartPool.sol

```
    function addAdmin(address admin) external onlyOwner{
        ...
    }
```

```
    function addAdmins(address[] calldata admins) external onlyOwner{
        ...
    }

   function setOperator(address newOperator) external onlyOwner {
        ...
    }

    function rebalanceOut(address token, uint256 amount) external whenNotPaused
nonReentrant onlyOperator() {
        ...
    }

    function swapOut(
        address token,
        uint256 amount,
        address swapTarget,
        address spender,
        bytes calldata swapCallData,
        address receiver,
        bytes32 hashlock,
        bytes calldata preimage
    ) external whenNotPaused nonReentrant onlyAdmin {
        ...
    }
```

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 the owner and operate roles are managed by a multi-signature wallet, and the admin is managed by the platform's EOA address.

[N6] [Suggestion] Expired failed orders cannot be refunded

Category: Design Logic Audit

Content

In the UxuySmartPool contract, the swapConfirm function is used by the Admin to confirm whether the order status is successful. If the order execution fails, a refund needs to be made to the user. However, one of the checks requires the current timestamp to be between `order.timelock` and `order.timelock + 1 days`. This means that if, for some reasons such as network congestion on the chain, the current time exceeds `order.timelock + 1 days`, it will be impossible to refund the user for the failed order.

Code Location:

ethereum/contracts/UxuySmartPool.sol#L204

```
   function swapConfirm(bytes32 hashlock, bool success) external whenNotPaused
nonReentrant onlyAdmin {
        require(orders[hashlock].sender != address(0), "Order not exist");
        SwapOrder storage order = orders[hashlock];
        require(!order.confirmed, "Already handled");

        if(success) {
            order.confirmed = true;
            emit SwapConfirmed(hashlock, order.sender, order.token, order.amount);
        } else {
            require(block.timestamp > order.timelock, "Timelock not expired");
            require(block.timestamp <= order.timelock + 1 days, "Confirmation window
passed");

            order.confirmed = true;
            order.refunded = true;

            IERC20(order.token).safeTransfer(order.sender, order.amount);
            emit SwapRefunded(hashlock, order.sender, order.token, order.amount);
        }
    }
```

Solution

It is recommended to remove the upper-limit check of the expiration time, and instead, allow the refund to be made as long as the current time exceeds order.timelock.

Status

Acknowledged; The project team responded that in order to safeguard against malicious refunds, only a one-day validity period is set. After the validity period, off-chain handling can be carried out.

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002507010002	SlowMist Security Team	2025.07.01 - 2025.07.01	Medium 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 risks and 3 suggestion. All the findings were acknowledged. Since the permissions of the core roles are still managed by an EOA, the risk level of the report remains medium for the time being.

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