Smart Contract Security Audit Report

Table Of Contents

1 Executive Summary

On 2025.06.27, the SlowMist security team received the OURO Protocol team's security audit application for OURO Protocol Phase2, 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

OURO Protocol is a dual-token DeFi ecosystem featuring USDO, a collateralized stablecoin, and STAR, the platform's reward-bearing token. Users accumulate STAR primarily by paying a transaction tax on USDO transfers or by directly purchasing it through authorized Minters. Held over time, STAR is allocated into compounding reward epochs, entitling users to generate a passive income in USDO. These rewards are claimed via a unique two-step process where users first apply for progressively larger airdrops and then claim the USDO in a subsequent epoch. The protocol is governed by a dynamic epoch system and includes referral and VIP programs, all designed to reward long-term participation and foster a sustainable, user-driven economy.

3.2 Vulnerability Information

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

NO	Title	Category	Level	Status
N1	Risk of excessive authority	Authority Control Vulnerability Audit	Medium	Acknowledged
N2	Unsafe Token Reactivation Without Price Validation	Design Logic Audit	Medium	Fixed
N3	Missing Stale Price Check for Chainlink Oracle	Design Logic Audit	Medium	Fixed
N4	Asset Value is Ignored During Deposit and Redeem, Allowing Value Extraction	Design Logic Audit	Medium	Acknowledged
N5	Reward System Paralysis if Owner Fails to Initialize Rates	Design Logic Audit	Low	Acknowledged
N6	Missing the event record	Others	Suggestion	Fixed
N7	Preemptive Initialization	Race Conditions Vulnerability	Suggestion	Acknowledged
N8	Epoch Advancement Relies on User-Triggered	Design Logic Audit	Information	Acknowledged
N9	Potential Gas Exhaustion DoS in Reward Calculation	Denial of Service Vulnerability	Information	Acknowledged
N10	Airdrop Level Hard Cap May Cause User Exclusion in Edge Cases	Others	Information	Acknowledged
N11	USDO Token Compatibility Reminder	Others	Information	Acknowledged

4 Code Overview

4.1 Contracts Description

Audit Version:

https://github.com/ouro-org/contract

commit: 5be554c0c6ba8e490dd3c4c67b2cf41f5010f9af

FIxed Version:

https://github.com/ouro-org/contract

commit: 877d71c1b77a521bfcf32ea0768b0309ebf5141f

```
./contracts
├── OURODataLib.sol
├──OUROGenesisVip.sol
├──OUROMintRouter.sol
├──OUROMinterManager.sol
├──OURONormalVip.sol
├──OUROProtocolProxy.sol
├──OUROProtocolUpgradeable.sol
├──OUROTokenRegistry.sol
├──OUROTreasure.sol
├──OUROVipManager.sol
├──USDO.sol
├──STAR.sol
├──interface
├──structs
```

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:

OUROGenesisVip			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	ERC721 Ownable
mint	Public	Can Modify State	onlyVipManager

OUROGenesisVip			
setVipManager	External	Can Modify State	onlyOwner
setBaseURI	Public	Can Modify State	onlyOwner
_baseURI	Internal	-	-
_update	Internal	Can Modify State	-

STAR			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	ERC20 Ownable
decimals	Public	-	-
mint	External	Can Modify State	onlyOuroProtocol
_update	Internal	Can Modify State	-

USDO			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	ERC20 Ownable
decimals	Public	-	-
mint	External	Can Modify State	onlyOuroProtocol
burn	Public	Can Modify State	onlyOuroProtocol
_update	Internal	Can Modify State	-
update	External	Can Modify State	onlyOuroProtocol
setConsumerTaxExemptReceiver	External	Can Modify State	onlyRole
setConsumerTaxExemptSender	External	Can Modify State	onlyRole

OUROVipManager			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	Ownable
joinNormalMemberShip	External	Can Modify State	-
joinGenesisMemberShip	External	Can Modify State	-
isNormalVip	Public	-	-
isGenesisVip	Public	-	-
isVip	External	-	-
withdrawVipIncome	External	Can Modify State	onlyRole

OURONormalVip			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	ERC721 Ownable
setVipManager	External	Can Modify State	onlyOwner
mint	Public	Can Modify State	onlyVipManager
setBaseURI	Public	Can Modify State	onlyOwner
_baseURI	Internal	-	-
_update	Internal	Can Modify State	-

OUROProtocolUpgradeable			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	-
initialize	Public	Can Modify State	initializer
_authorizeUpgrade	Internal	Can Modify State	onlyRole

OUROProtocolUpgradeable			
initializeCore	External	Can Modify State	onlyOwner
deposit	External	Can Modify State	nonReentrant whenNotPaused
_handleReferrer	Private	Can Modify State	-
redeem	External	Can Modify State	nonReentrant whenNotPaused
handleConsumerTax	External	Can Modify State	nonReentrant
_calculateReferrerRebate	Private	Can Modify State	-
_recordUserTaxBurned	Private	Can Modify State	-
_appendStarToPendingEpoch	Private	Can Modify State	-
_generateNewEpoch	Private	Can Modify State	-
_validStarStartAmount	Private	-	-
_checkAndAdvanceCurrentEpoch	Private	Can Modify State	-
_calculateEpochDelayRate	Private	-	-
getEpochInfo	External	-	-
userCumulativeApplyableRewardsAmount	External	-	-
_userCumulativeRewardsAmountAtEpoch	Private	-	-
suggestApplyEpoch	External	-	-
applyAirdrop	External	Can Modify State	nonReentrant whenNotPaused
currentEpochPendingAmount	External	-	-
calculateClaimableAmount	External	-	-

OUROProtocolUpgradeable			
_checkPendingApplyRewards	Private	Can Modify State	-
claimAirdrop	External	Can Modify State	nonReentrant whenNotPaused
userNextApplyAmount	Public	-	-
calculateApplyAmount	Public	-	-
userCanBeReferrer	Public	-	-
getUserValidStar	External	-	-
setReferrerQualification	External	Can Modify State	onlyRole
mintStarByMinter	External	Can Modify State	-
withdrawRedeemTax	External	Can Modify State	onlyRole whenNotPaused
addConsumerTaxExemptSenderAddress	External	Can Modify State	onlyRole
removeConsumerTaxExemptSenderAddress	External	Can Modify State	onlyRole
setCumulativeValues	External	Can Modify State	onlyOwner
_getCumulativeRate	Private	-	-
_getCumulativeRateByEpoch	Private	-	-
pause	External	Can Modify State	onlyRole
unpause	External	Can Modify State	onlyRole

OUROTreasure			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	Ownable
withdraw	External	Can Modify State	onlyOuroProtocol whenNotPaused

OUROTreasure			
balanceOf	External	-	-
pause	External	Can Modify State	onlyRole
unpause	External	Can Modify State	onlyRole

OUROMinterManager			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	Ownable
addMinter	External	Can Modify State	onlyRole
updateMinterName	External	Can Modify State	onlyRole nonReentrant
removeMinter	External	Can Modify State	onlyRole nonReentrant
getMintersInfoAndAllowance	External	-	-
mintByMinter	External	Can Modify State	onlyOuroProtocol whenNotPaused
removeDuplicates	Internal	-	-
pause	External	Can Modify State	onlyRole
unpause	External	Can Modify State	onlyRole

OUROTokenRegistry			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	Ownable
addToken	External	Can Modify State	onlyRole
removeToken	External	Can Modify State	onlyRole
reactivateToken	External	Can Modify State	onlyRole
updateRedeemFee	External	Can Modify State	onlyRole

OUROTokenRegistry			
activatePendingFee	External	Can Modify State	onlyRole
activatePendingToken	External	Can Modify State	onlyRole
getRedeemFee	External	-	-
getPendingFee	External	-	-
getActivationTime	External	-	-
isSupported	External	-	-
getLastUpdated	External	-	-
getBaseRedeemFeeRate	External	-	-
getPendingTokenInfo	External	-	-
isERC20	Internal	-	-
setPriceFeed	External	Can Modify State	onlyRole
getTokenPriceUsd	Public	-	-
_checkTokenPrice	Internal	-	-

OUROMintRouter			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	Ownable
sendCommissionToMinter	External	Can Modify State	onlyOuroProtocol

OUROProtocolProxy			
Function Name	Visibility	Mutability	Modifiers
<Constructor>	Public	Can Modify State	ERC1967Proxy

4.3 Vulnerability Summary

[N1] [Medium] Risk of excessive authority

Category: Authority Control Vulnerability Audit

Content

1.In the OUROProtocolUpgradeable contract, the owner role can set or remove the referrer address, the transferBurnExemptSenders address, abd the consumerTaxExemptSenders address arbitrarily.

Code location:

contracts/OUROProtocolUpgradeable.sol#L670-L675, L722-L728

```
    function setReferrerQualification(address referrer) external
onlyRole(DEFAULT_ADMIN_ROLE) override {
        ...
    }

    function addConsumerTaxExemptSenderAddress(address addr) external
onlyRole(DEFAULT_ADMIN_ROLE) {
        ...
    }

    function removeConsumerTaxExemptSenderAddress(address addr) external
onlyRole(DEFAULT_ADMIN_ROLE) {
        ...
    }
```

2.In the OUROMinterManager contract, the EFAULT_ADMIN_ROLE can add or remove minters arbitrarily, and set the dailyAllowance daliy mint amount for the minter.

contracts/OUROMinter.sol#L51-74, L82-L88

```
    function addMinter(
        address minter,
        string memory name,
        uint256 dailyAllowance
    ) external onlyRole(DEFAULT_ADMIN_ROLE) override {
        ...
    }

    function removeMinter(address minter) external onlyRole(DEFAULT_ADMIN_ROLE)
nonReentrant override {
        ...
    }
```

3.The DEFAULT_ADMIN_ROLE and REGISTRAR_ROLE in the OUROTokenRegistry contract can call the addToken() and removeToken() functions to list or de-list a token. Furthermore, this role can change a token's price feed oracle via setPriceFeed() or bypass safety delays by reactiving a risky token via reactivateToken() function.

Code location:

contracts/OUROTokenRegistry.sol#L57-L98 ,L201-L205

```
   function addToken(address token, uint256 fee) external onlyRole(REGISTRAR_ROLE)
override {
        ...
    }

    function removeToken(address token) external onlyRole(REGISTRAR_ROLE) override {
        ...
    }

    function reactivateToken(address token) external onlyRole(REGISTRAR_ROLE)
override {
        ...
    }

   function setPriceFeed(address token, address aggregator) external
onlyRole(REGISTRAR_ROLE) {
        ...
    }
```

4.The PAUSER_ROLE of the OUROTreasure can pause the withdraw process from the OuroProtocol. This may cause the suspension of the agreement to be inconsistent with the suspension of Treasure.

contracts/OUROTreasure.sol#L52-L54

```
  function pause() external onlyRole(PAUSER_ROLE) {
        _pause();
    }
```

5.The DEFAULT_ADMIN_ROLE can set the consumerTaxExemptReceivers and consumerTaxExemptSenders addresses arbitrarily through the setConsumerTaxExemptReceiver and setConsumerTaxExemptSender functions to bypass the transfer tax.

contracts/USDO.sol#L72-L80

```
    function setConsumerTaxExemptReceiver(address receiver, bool isExempt) external
onlyRole(DEFAULT_ADMIN_ROLE) {
        ...
    }

    function setConsumerTaxExemptSender(address sender, bool isExempt) external
onlyRole(DEFAULT_ADMIN_ROLE) {
        ...
    }
```

6.The UUPSUpgradeable owner relevant authority can upgrade the contract, leading to the risk of over-privileged in this role.

Code location:

contracts/OUROProtocolUpgradeable.sol#L163

```
    function _authorizeUpgrade(address newImplementation) internal override
onlyRole(UPGRADER_ROLE) {}
```

Solution

In the short term, transferring owner ownership 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. And the authority involving user funds should be managed by the community, and the EOA address can manage the authority involving emergency contract suspension. This ensures both a quick response to threats and the safety of user funds.

Status

Acknowledged

[N2] [Medium] Unsafe Token Reactivation Without Price Validation

Category: Design Logic Audit

Content

The reactivateToken() function in the OUROTokenRegistry contract allows a REGISTRAR_ROLE to re-enable a previously deactivated token by simply setting its isActive flag to true. The function fails to re-run the critical _checkTokenPrice validation or any other checks that are performed when a token is first added or not added. This

could allow an administrator to re-list a token that was delisted for a valid reason such as de-pegging from its $1 value or list a unadded token by bypassing the `!tokens[token].isActive` check and default true return value of the _checkTokenPrice function. Malicious actors could then use this re-listed/unlisted "bad" token in the main protocol at an incorrect value, potentially leading to a drain of protocol funds.

Code location:

contracts/OUROTokenRegistry.sol#L94-L98

```
    function reactivateToken(address token) external onlyRole(REGISTRAR_ROLE) override
{
        require(!tokens[token].isActive, "Token already active");
        tokens[token].isActive = true;
        emit TokenReactivated(token);
     }
```

Solution

It is recommended to enforce security checks within the reactivateToken function, especially the _checkTokenPrice(token) validation and check the token is listed by the lastUpdateTime.

Status

Fixed; The project team stated that the default return for the _checkTokenPrice function is expected.

[N3] [Medium] Missing Stale Price Check for Chainlink Oracle

Category: Design Logic Audit

Content

The getTokenPriceUsd() function in the OUROTokenRegistry contract fetches the latest price from a Chainlink V3 Aggregator but fails to validate the updatedAt timestamp returned by the latestRoundData() call. This could lead the contract to use a stale price that has not been updated for a significant period (e.g., hours or days) due to oracle downtime or network issues. Using an outdated price for critical security checks, such as validating a stablecoin's peg, is a significant risk and can undermine the integrity of the token listing process.

Code location:

contracts/OUROTokenRegistry.sol#L207-L217

```
    function getTokenPriceUsd(address token) public view returns (uint256 price, uint8
decimals) {
```

```
        address aggregator = priceFeedRegistry[token];
        require(aggregator != address(0), "No aggregator for token");

        AggregatorV3Interface priceFeed = AggregatorV3Interface(aggregator);
        (, int256 rawPrice, , , ) = priceFeed.latestRoundData();
        require(rawPrice > 0, "Invalid price");

        decimals = priceFeed.decimals();
        price = uint256(rawPrice);
    }
```

Solution

It is recommended to check the returned updatedAt timestamp from the latestRoundData call to ensure the price data is recent.

Status

Fixed

[N4] [Medium] Asset Value is Ignored During Deposit and Redeem, Allowing Value Extraction

Category: Design Logic Audit

Content

The protocol's deposit and redeem mechanism ignores the actual market value of the assets involved, relying solely on their decimal precision for conversion. The deposit() function converts any supported token to USDO based on a simple decimal adjustment, effectively treating all supported tokens as having a 1:1 value with USDO. The redeem() function compounds this issue by allowing a user to deposit one asset (e.g., a de-pegged stablecoin worth $0.50) and then redeem their USDO for a different, high-value asset (e.g., USDC worth $1.00). This creates a direct and easily exploitable arbitrage opportunity, allowing an attacker to drain the protocol of its most valuable assets.

Attack Scenario:

The OUROTokenRegistry lists two tokens: USDC (worth $1.00) and a de-pegged stablecoin, BAD (worth $0.10).

An attacker deposits 1,000,000 BAD tokens. The deposit function incorrectly mints approximately 1,000,000 USDO for the attacker, assuming BAD is worth $1.00.

The attacker immediately calls redeem(), using their 1,000,000 USDO to withdraw USDC.

The attacker successfully withdraws nearly 1,000,000 USDC from the treasure, having only deposited assets worth $100,000.

The attacker repeats this process, draining the protocol of all its valuable assets.

Code location:

contracts/OUROProtocolUpgradeable.sol#L192-L206, L247-L263

```
    function deposit(uint256 amount, address referrer, address token) external
nonReentrant whenNotPaused override {
        ...
        uint256 usdoAmount = OURODataLib.convert(amount,
IERC20MetadataUpgradeable(token).decimals(),
IERC20MetadataUpgradeable(usdo).decimals());
        userInfos[msg.sender].totalDepositAmount += usdoAmount;
        ...
    }

    function redeem(uint256 amount, address token) external nonReentrant whenNotPaused
override {
        ...
        uint256 tokenAmount = OURODataLib.convert(amount,
IERC20MetadataUpgradeable(usdo).decimals(),
IERC20MetadataUpgradeable(token).decimals());
        uint256 redeemTax = tokenAmount *
IOUROTokenRegistry(tokenRegistry).getRedeemFee(token) /
IOUROTokenRegistry(tokenRegistry).getBaseRedeemFeeRate();
        uint256 redeemAmount = tokenAmount - redeemTax;
        ...
    }
```

Solution

It's recommended to track user deposits and restrict redemption.

Status

Acknowledged; The project team stated that they would **only** support USDT as the Deposit token currently.

[N5] [Low] Reward System Paralysis if Owner Fails to Initialize Rates

Category: Design Logic Audit

Content

The entire airdrop reward system is critically dependent on the owner calling the setCumulativeValues() function to populate the cumulativeValues array. If the owner neglects to perform this one-time setup, the array's length will remain zero. Consequently, any call to _getCumulativeRate, which is a core part of the reward calculation pipeline

(applyAirdrop -> _userCumulativeRewardsAmountAtEpoch -> _getCumulativeRateByEpoch -> _getCumulativeRate), will fail due to the require(index < cumulativeValues.length) check. This will cause all applyAirdrop transactions to revert with an "Index out of bounds" error, effectively paralyzing the reward distribution mechanism and preventing any user from ever applying for their earned rewards.

Code location:

contracts/OUROProtocolUpgradeable.sol#L713-L720

```
function setCumulativeValues(uint[] calldata _values) external onlyOwner() {
     require(_values.length == 100, "Must be 100 entries");
     require(cumulativeValues.length == 0, "Cumulative values already set");
     for (uint i = 0; i < 100; i++) {
         cumulativeValues.push(_values[i]);
     }
     emit CumulativeValuesSet(cumulativeValues);
 }
```

Solution

It is recommended to add a check at the beginning of the reward calculation process. A require statement should be added to the start of the _userCumulativeRewardsAmountAtEpoch function to validate that the rates have been initialized.

Status

Acknowledged

[N6] [Suggestion] Missing the event record

Category: Others

Content

The following functions are missing event records after calling the modification.

Code location:

contracts/OUROGenesisVip.sol#L35-L37

contracts/OURONormalVip.sol#L35-L37

```
function setBaseURI(string memory baseURI) public onlyOwner() {
     _baseTokenURI = baseURI;
```

```
    }
```

Solution

It is recommended to record events when sensitive parameters are modified for self-inspection or community review.

Status

Fixed

[N7] [Suggestion] Preemptive Initialization

Category: Race Conditions Vulnerability

Content

By calling the initialize function to initialize the contract, there is a potential issue that malicious attackers can preemptively call the initialize function to initialize.

Code location:

contracts/OUROProtocolUpgradeable.sol#L131-L161

```
function initialize(
    ...
) public initializer {
    ...
}
```

Solution

It is suggested that the initialization operation can be called in the same transaction immediately after the contract is created to avoid being maliciously called by the attacker.

Status

Acknowledged

[N8] [Information] Epoch Advancement Relies on User-Triggered

Category: Design Logic Audit

Content

The protocol's time-based epoch (currentEpochNumber) does not advance automatically. Progression depends on the _checkAndAdvanceCurrentEpoch function being called, which is triggered by specific user-initiated transactions,

namely handleConsumerTax and mintStarByMinter. During periods of low or no user activity, the epoch clock could stall even if its designated time duration has passed. However, the practical risk of prolonged stagnation is low, as users are economically incentivized to trigger these functions to make their own rewards claimable. The system is designed to catch up quickly once activity resumes.

contracts/OUROProtocolUpgradeable.sol#L265-L315, L659-L689

```
    function handleConsumerTax(address from, address to, uint256 amount) external
nonReentrant override returns (uint256)  {
        ...
    }

    function mintStarByMinter(address minterAddress, uint256 starAmount) external
override {
        ...
    }
```

Solution

For enhanced future robustness, a dedicated, externally callable "keep-alive" function or an automated solution could be considered, but it is not critical for the current design.

Status

Acknowledged

[N9] [Information] Potential Gas Exhaustion DoS in Reward Calculation

Category: Denial of Service Vulnerability

Content

The function _userCumulativeRewardsAmountAtEpoch, which is called by applyAirdrop, calculates a user's total rewards by iterating through historical epochs. Although an optimization exists to limit the look-back period, this loop can still run up to approximately 100 times. For a user who has been active for a long time but has not claimed rewards recently, calling applyAirdrop could create a transaction with a gas cost so high that it exceeds the block gas limit. This would cause the transaction to consistently fail, effectively preventing the user from applying for their earned rewards and creating a self-inflicted Denial of Service (DoS) situation.

Code location:

contracts/OUROProtocolUpgradeable.sol#L481-L526

```
    function _userCumulativeRewardsAmountAtEpoch(uint32 applyEpoch) private view
returns (uint256) {
        ...
        uint256 totalRewards = 0;
        uint32 startEpoch = 1;
        if (userRewardsInfo.lastApplyEpoch > 100) {
            startEpoch = userRewardsInfo.lastApplyEpoch - 100;
        }
        ...
    }
```

Solution

It is recommended to enforce an on-chain limit for the calculation range. Add a require statement at the beginning of the applyAirdrop function to ensure the difference between applyEpoch and the user's lastApplyEpoch does not exceed a provably safe number (e.g., 50). This forces users to claim rewards in smaller, manageable batches, completely mitigating the risk of transaction failure due to gas exhaustion.

Status

Acknowledged; The team stated that they use suggestApplyEpoch function to remind users on the front end, suggesting and letting users apply for rewards through a maximum of 50 epochs.

[N10] [Information] Airdrop Level Hard Cap May Cause User Exclusion in Edge Cases

Category: Others

Content

The calculateApplyAmount function contains a require(level < 64) check, creating a hard cap for a user's airdrop level at 63. While this creates a theoretical maximum, reaching this limit is practically impossible. A user would first need to accumulate rewards equivalent to 2^64 - 1 base units of USDO (an astronomical sum) to be able to apply for all levels up to 63. The exponential growth of the required rewards creates a natural economic barrier long before any user could approach the technical limit.

Code location:

OUROProtocolUpgradeable.sol#L639-L643

```
    function calculateApplyAmount(uint64 level) public view returns (uint256) {
        require(level < 64, "Level too high");
        uint256 claimable = 1 << level;
```

```
        return claimable * 10 ** IERC20MetadataUpgradeable(usdo).decimals();
    }
```

Solution

It’s recommended to add a level reset mechanism that allows high-level users to reset their level at a cost. Or remove the hard-coded level 64 limit or implement a more sustainable growth model.

Status

Acknowledged

[N11] [Information] USDO Token Compatibility Reminder

Category: Others

Content

The USDO token implements non-standard ERC20 behavior that deviates from typical token transfer mechanics. During normal transfer operations, USDO automatically applies a tax mechanism that burns a portion of the transferred amount and generates STAR tokens for users. This non-standard behavior may cause compatibility issues when integrating with external protocols, decentralized exchanges, or other smart contracts that expect standard ERC20 token behavior.

External protocols typically assume that when transferring X tokens, exactly X tokens will be received by the recipient. However, USDO’s tax mechanism means that the actual amount received will be less than the amount sent (after tax deduction), and additional side effects occur (STAR token generation, referrer rewards, epoch allocations). This can lead to unexpected behavior in:

Automated Market Makers (AMMs) and decentralized exchanges

Lending and borrowing protocols

Multi-token operations expecting precise balance calculations

Smart contracts that don't account for deflationary token mechanics

Integration with standard DeFi protocols expecting vanilla ERC20 behavior

Developers and users should be aware of these non-standard characteristics when integrating USDO with external systems or protocols.

Solution

N/A

Status

Acknowledged

5 Audit Result

Audit Number	Audit Team	Audit Date	Audit Result
0X002506300001	SlowMist Security Team	2025.06.27 - 2025.06.30	Medium Risk

Summary conclusion: The SlowMist security team uses a manual and the SlowMist team's analysis tool to audit the project. During the audit work, we found 4 medium risk, 1 low risk, 2 suggestions, and 4 information. All the findings were fixed or acknowledged. The code was not deployed to the mainnet.

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