CLASH OF COINS CLAIM SMART CONTRACTS SECURITY AUDIT REPORT

OXORIO

DECEMBER 25, 2024

CONTENTS

1. AUDIT OVERVIEW 4

1.1. PROJECT BRIEF 5

1.2. PROJECT TIMELINE 6

1.3. AUDITED FILES 7

1.4. PROJECT OVERVIEW 8

1.5. CODEBASE QUALITY ASSESSMENT 9

1.6. SUMMARY OF FINDINGS 11

1.7. CONCLUSION 13

2. FINDINGS REPORT 14

2.1. CRITICAL 15

2.2. MAJOR 16

2.3. WARNING 17

W-01 msg.sender address usage as owner in ClaimSoftCurrency, RetroDropWithMerkle 17

W-02 nonce can be reused with different points in ClaimSoftCurrency 18

W-03 Missing condition that points is not zero in RetroDropWithMerkle 19

W-04 Minting only points or only coins is not allowed in ClaimRewards 20

2.4. INFO 21

I-01 Solidity version can be specified as 0.8.27 21

I-02 Missing record in RetroDropWithMerkle 22

I-03 Specify the user instead of msg.sender as a separate function parameter in RetroDropWithMerkle, ClaimSoftCurrency 23

I-04 Incorrect error message in ClaimSoftCurrency 24

I-05 Replace require with custom errors 25

I-06 Incomplete description in the README file 26

I-07 Redundant error in ClaimSoftCurrency 27

I-08 Rename claimedPoints to pointClaimed in RetroDropWithMerkle 28

I-09 Missing error descriptions in ClaimSoftCurrency, RetroDropWithMerkle 29
I-10 Unnecessary <= 0 condition for uint in ClaimRewards 30
3. DEPLOY VERIFICATION 31
3.1. SCOPE 32
3.2. CONCLUSION 33
3.3. VERIFICATION 34
Network specific behavior 34
Scope checking 34
Audit report investigation 34
Deploy script check 35
Deployment verification 35
Initialization parameters check 36
Role model verification 36
Storage Check 36
Documentation Verification 37
4. APPENDIX 38
4.1. DISCLAIMER 39
4.2. SECURITY ASSESSMENT METHODOLOGY 40
4.3. CODEBASE QUALITY ASSESSMENT REFERENCE 42
Rating Criteria 43
4.4. FINDINGS CLASSIFICATION REFERENCE 44
Severity Level Reference 44
Status Level Reference 44
4.5. ABOUT OXORIO 46

1 AUDIT OVERVIEW

1.1 PROJECT BRIEF

Title	Description
Client	Clash of Coins
Project name	Clash of Coins claim contracts
Category	Gaming
Website	https://clashofcoins.com/
Repository	https://github.com/onewayblock/coc-claim-contracts
Documentation	https://github.com/onewayblock/coc-claim-contracts/blob/main/README.md
Initial Commit	63918d3513f13a6311cf3519b1b59c03682a0c1d
Final Commit	a8facc20dc1097f27a2463ddf9e2e530f12ceb29
Platform	L2
Network	Base
Languages	Solidity
Lead Auditor	Alexander Mazaletskiy - am@oxor.io
Project Manager	Nataly Demidova - nataly@oxor.io

1.2 PROJECT TIMELINE

The key events and milestones of the project are outlined below.

Date	Event
December 10, 2024	Client approached Oxorio requesting an audit.
December 12, 2024	The audit team commenced work on the project.
December 13, 2024	Submission of the preliminary report.
December 19, 2024	Submission of the comprehensive report.
December 23, 2024	Client feedback on the report was received.
December 25, 2024	Submission of the final report incorporating client's verified fixes.

1.3 AUDITED FILES

The following table contains a list of the audited files. The scc tool was used to count the number of lines and assess complexity of the files.

	File	Lines	Blanks	Comments	Code	Complexity
1	contracts/ClaimSoftCurrency.sol	91	16	31	**44**	9
2	contracts/RetroDropWithMerkle.sol	79	13	27	**39**	3
	Total	**170**	**29**	**58**	**83**	**6**

Lines: The total number of lines in each file. This provides a quick overview of the file size and its contents.

Blanks: The count of blank lines in the file.

Comments: This column shows the number of lines that are comments.

Code: The count of lines that actually contain executable code. This metric is essential for understanding how much of the file is dedicated to operational elements rather than comments or whitespace.

Complexity: This column shows the file complexity per line of code. It is calculated by dividing the file's total complexity (an approximation of cyclomatic complexity that estimates logical depth and decision points like loops and conditional branches) by the number of executable lines of code. A higher value suggests greater complexity per line, indicating areas with concentrated logic.

1.4 PROJECT OVERVIEW

The ClashOfCoins project implements a series of smart contracts designed to facilitate claim-based mechanisms for an on-chain gaming ecosystem. The core functionality revolves around enabling users to claim rewards based on predefined criteria and interact with the platform's token economy. The contracts are written in Solidity and are optimized for deployment on the Base blockchain. Key features include reward distribution logic, ownership management, and mechanisms to ensure the integrity and security of the claiming process. The contracts also leverage access control patterns to maintain operational security.

The primary goal of the project is to establish a transparent and efficient claim and reward system for users participating in the ClashOfCoins game. Key functionalities include managing reward allocation, validating user eligibility, and integrating token-based interactions within the ecosystem. Notable architectural features include role-based access controls and mechanisms to mitigate common smart contract vulnerabilities, ensuring the system's robustness and reliability in a decentralized environment.

1.5 CODEBASE QUALITY ASSESSMENT

The Codebase Quality Assessment table offers a comprehensive assessment of various code metrics, as evaluated by our team during the audit, to gauge the overall quality and maturity of the project's codebase. By evaluating factors such as complexity, documentation and testing coverage to best practices, this table highlights areas where the project excels and identifies potential improvement opportunities. Each metric receives an individual rating, offering a clear snapshot of the project's current state, guiding prioritization for refactoring efforts, and providing insights into its maintainability, security, and scalability. For a detailed description of the categories and ratings, see the Codebase Quality Assessment Reference section.

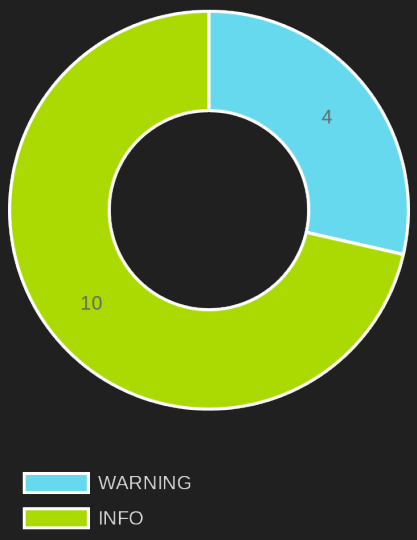
Category	Assessment	Result
Access Control	The project's codebase implements a robust access control mechanism with multiple differentiated roles to manage system functionalities efficiently.	**Excellent**
Arithmetic	The smart contract code does not contain any arithmetic operations.	**Not Applicable**
Complexity	The code is consistent, and follows best practices in coding styles, making it easy to comprehend and review.	**Excellent**
Data Validation	The project ensures data validation across all components.	**Excellent**
Decentralization	The project does not incorporate a decentralized approach to management, and therefore, the metric is not applicable in this context.	**Not Applicable**
Documentation	The project documentation covers all components, is up-to-date, and is centralized in a single source.	**Excellent**
External Dependencies	The project does not interact with any external smart contracts in its logic; therefore, this metric is not applicable in this context.	**Not Applicable**
Error Handling	The project demonstrates robust exception handling throughout the codebase. Custom errors with clear naming and descriptions are used.	**Excellent**
Logging and Monitoring	The project exhibits excellent logging capabilities, recording all important events within the system.	**Excellent**

Category	Assessment	Result
Low-Level Calls	The project is free from low-level calls, ensuring a higher level of security by avoiding potential pitfalls associated with direct, low-level interactions with the blockchain.	**Not Applicable**
Testing and Verification	The codebase exhibits commendable test coverage, demonstrating a strong commitment to verifying functionality and reliability.	**Excellent**

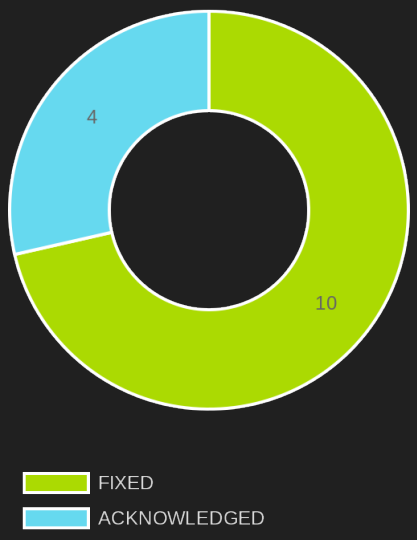
1.6 SUMMARY OF FINDINGS

The table below provides a comprehensive summary of the audit findings, categorizing each by status and severity level. For a detailed description of the severity levels and statuses of findings, see the Findings Classification Reference section.

Severity	TOTAL	NEW	FIXED	ACKNOWLEDGED	NO ISSUE
CRITICAL	**0**	0	0	0	0
MAJOR	**0**	0	0	0	0
WARNING	**4**	0	3	1	0
INFO	**10**	0	7	3	0
TOTAL	**14**	**0**	**10**	**4**	**0**



Issue distribution by severity



Issue distribution by status

This table provides an overview of the findings across the audited files, categorized by severity level. The table enables to quickly identify areas that require immediate attention and prioritize remediation efforts accordingly.

File	TOTAL	CRITICAL	MAJOR	WARNING	INFO
contracts/ClaimSoftCurrency.sol	**5**	0	0	2	3

File	TOTAL	CRITICAL	MAJOR	WARNING	INFO
contracts/RetroDropWithMerkle.sol	**5**	0	0	2	3
contracts/ClaimRewards.sol	**2**	0	0	1	1
contracts/ClaimSoftCurrency.sol	**2**	0	0	0	2
contracts/RetroDropWithMerkle.sol	**2**	0	0	0	2

1.7 CONCLUSION

Clash of Coins claim contracts have been audited, and no critical or major issues were found. However, a few recommendations were marked as warnings and informational. Some changes were proposed to follow best practices, reduce the potential attack surface, simplify code maintenance, and improve readability. No severe attack vectors or broken features were identified.

All identified issues have been appropriately fixed or acknowledged by the client, so the contracts are deemed secure to use according to our security criteria. The final commit identifier with all fixes is a8facc20dc1097f27a2463ddf9e2e530f12ceb29. This version is recommended for deployment and further system testing.

2 FINDINGS REPORT

OXORIO

2.1 CRITICAL

No critical issues found.

2.2 MAJOR

No major issues found.

2.3 WARNING

W-01	`msg.sender` address usage as owner in `ClaimSoftCurrency`, `RetroDropWithMerkle`
Severity	**WARNING**
Status	• FIXED

Location

File	Location	Line
ClaimSoftCurrency.sol	contract `ClaimSoftCurrency` > `constructor`	35
RetroDropWithMerkle.sol	contract `RetroDropWithMerkle` > `constructor`	26

Description

In the mentioned locations, the `msg.sender` address is used as the owner for a newly deployed contract. From a best practice perspective, it is not advisable to transfer the `msg.sender` dispatcher address as the owner, as the deployer is often a hot wallet.

Recommendation

We recommended specifying the owner as a separate parameter `owner` and ensuring a non-zero address, which is already checked in the `Ownable` contract.

Update

Fixed at `15865bdf3e778674be35e2a422c8e83656544efd`

W-02	`nonce` can be reused with different `points` in `ClaimSoftCurrency`
Severity	**WARNING**
Status	• FIXED

Location

File	Location	Line
ClaimSoftCurrency.sol	contract `ClaimSoftCurrency` > function `claimPoints`	47

Description

In the function `claimPoints` of the contract `ClaimSoftCurrency`, it is possible to use the same `nonce` with different `points`. In this case, the `messageHash` will differ, but the `nonce` will remain the same, allowing potential re-use.

Recommendation

We recommended not using `nonce` as a function parameter but instead managing it as part of the contract. Specifically, add a `mapping(address => uint256) public nonces;` and increment the nonce for each claim, like so: `nonces[account]++`. This ensures that each nonce is unique and sequential, preventing attempts to reuse the same nonce.

This approach makes each transaction unique with its own `nonce`, which avoids redundancy. The `messageHash` will also naturally be different for each user action, and the following design ensures it cannot be re-executed:

```
require(!executedHashes[messageHash], "Tx already executed");
executedHashes[messageHash] = true;
```

Update

Fixed at `15865bdf3e778674be35e2a422c8e83656544efd`

W-03	Missing condition that `points` is not zero in `RetroDropWithMerkle`
Severity	**WARNING**
Status	• FIXED

Location

File	Location	Line
RetroDropWithMerkle.sol	contract `RetroDropWithMerkle` > function `isParticipating`	70

Description

In the function `isParticipating` of the contract `RetroDropWithMerkle`, there is no verification to ensure that `points` is non-zero, as required by the `claimPoints` function.

Recommendation

We recommended adding a check to ensure that the `points` variable value is not zero.

Update

Fixed at `15865bdf3e778674be35e2a422c8e83656544efd`

W-04	Minting only `points` or only `coins` is not allowed in `ClaimRewards`
Severity	**WARNING**
Status	• ACKNOWLEDGED

Location

File	Location	Line
ClaimRewards.sol	contract `ClaimRewards` > function `claimRewards`	70-76

Description

In the function `claimRewards` of the contract `ClaimRewards`, there are checks to ensure that the `points` and `coins` variables are not equal to zero:

```
if (points <= 0) {
  revert InvalidPoints();
}

if (coins <= 0) {
  revert InvalidCoins();
}
```

However, since both checks are mandatory, it is not possible to mint only points or only coins.

Recommendation

We recommend adding the ability to mint only points or only coins if this aligns with the business logic of the contract.

Update

Clash of Coins' Response

According to the product logic associated with the contract, the user can only receive both `coins` and `points` together. Scenarios where only one entity can be received are not provided for in the product. Therefore, at this stage, we will not make changes. However, we agree with the comment that a more flexible approach would be preferable.

2.4 INFO

I-01	Solidity version can be specified as `0.8.27`
Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
ClaimSoftCurrency.sol	-	2
RetroDropWithMerkle.sol	-	2

Description

In all mentioned locations, an unlocked compiler version is used (with `^`). This can lead to unintended errors if changes in minor compiler versions affect the smart contract logic.

Recommendation

We recommended using version `0.8.27` as specified in the Hardhat settings and locking the version by omitting the `^`.

Update

Fixed at `15865bdf3e778674be35e2a422c8e83656544efd`

I-02	Missing record in `RetroDropWithMerkle`
Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
RetroDropWithMerkle.sol	contract `RetroDropWithMerkle` > function `claimPoints`	44

Description

In the function `claimPoints` of the contract `RetroDropWithMerkle`, there is no record of how many `points` users have claimed.

Recommendation

We recommended adding a `mapping` to store information about the number of `points` each user has claimed and making it public so that it can always be checked on-chain.

Update

Fixed at `15865bdf3e778674be35e2a422c8e83656544efd`

I-03	Specify the user instead of `msg.sender` as a separate function parameter in `RetroDropWithMerkle`, `ClaimSoftCurrency`
Severity	**INFO**
Status	• ACKNOWLEDGED

Location

File	Location	Line
RetroDropWithMerkle.sol	contract `RetroDropWithMerkle` > function `claimPoints`	45
ClaimSoftCurrency.sol	contract `ClaimSoftCurrency` > function `claimPoints`	52

Description

In the mentioned locations, there is no separate function parameter that allows the sender to make a claim on behalf of any address.

Recommendation

We recommend specifying a separate function parameter instead of relying solely on `msg.sender`, for example, `account`. This will allow the sender to make a claim on behalf of any address.

Update

Clash of Coins' Response

We do not see a product requirement to allow claiming from other addresses, so we deliberately decided not to change this logic.

I-04	Incorrect error message in `ClaimSoftCurrency`
Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
ClaimSoftCurrency.sol	contract `ClaimSoftCurrency` > function `_verifySignature`	55

Description

In the function `_verifySignature` of the contract `ClaimSoftCurrency`, the error message is not entirely correct, as the `recover` function already includes a similar message:

```
require(_verifySignature(messageHash, signature), "Invalid signature");
```

Moreover, this message is not appropriate for the logic of this contract. In this contract, it actually signifies an invalid signer.

Recommendation

We recommended using the "Invalid signer" revert message instead of "Invalid signature."

Update

Fixed at `15865bdf3e778674be35e2a422c8e83656544efd`

I-05	Replace `require` with custom errors
Severity	**INFO**
Status	• FIXED

Description

Using `require` is considered an outdated method. To make the code more readable, provide clearer error messages, and improve debugging, it is recommended to use custom errors.

Note that the OpenZeppelin library already uses custom errors for this purpose.

Recommendation

We recommended replacing `require` statements with custom errors to enhance code clarity and maintainability.

Update

Fixed at `15865bdf3e778674be35e2a422c8e83656544efd`

I-06	Incomplete description in the `README` file
Severity	**INFO**
Status	• FIXED

Description

We identified the following inaccuracies in the `README` file:

1. Currently, there is no information on how to run tests and what dependencies are required. For example, I had to install the TypeScript dependency separately, and I selected the Node.js version according to the Hardhat version. Such details should be specified in the `package.json`, and a test command should be added to execute the tests. Basic environment information is necessary, especially if the repository is public.
2. Information about **Retrodrom** is superfluous, as there is no such contract in the repository.
3. The function signature `claimPoints(uint256 points, uint256 nonce, uint256 chainId, bytes memory signature)` should be changed to `claimPoints(uint256 points, bytes memory signature)`.
4. Add descriptions for errors.
5. Provide advanced information in deployment scripts for Hardhat ignition parameters (`./ignition/parameters.json`) and include a sample file:
6. `ClaimRewardsModule`: `owner`, `backendSigner`
7. `RetroDropWithMerkleModule`: `owner`, `merkleRoot`
8. Specify the Node.js version required to run tests and deployment.
9. Remove redundant networks (`ethereum-mainnet` and `ethereum-sepolia`).

Recommendation

We recommended addressing the inaccuracies listed above to improve the completeness and clarity of the `README` file.

Update

Fixed at `15865bdf3e778674be35e2a422c8e83656544efd` and `0c8f42e93d6f7a6a6fdd84252c109150d3371ba4`

I-07	Redundant error in `ClaimSoftCurrency`
Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
ClaimSoftCurrency.sol	contract `ClaimSoftCurrency`	27

Description

In the contract `ClaimSoftCurrency`, the error `TransactionAlreadyExecuted` is redundant and can be removed.

Recommendation

We recommended removing the redundant `TransactionAlreadyExecuted` error.

Update

Fixed at `0c8f42e93d6f7a6a6fdd84252c109150d3371ba4`

I-08	Rename `claimedPoints` to `pointClaimed` in `RetroDropWithMerkle`
Severity	**INFO**
Status	• ACKNOWLEDGED

Location

File	Location	Line
RetroDropWithMerkle.sol	contract `RetroDropWithMerkle`	16

Description

In the contract `RetroDropWithMerkle`, there is a `claimedPoints` storage variable. However, the name `pointClaimed` would be more appropriate, as it aligns with naming conventions used in other contracts and the event `PointClaimed`.

Recommendation

We recommend renaming the `claimedPoints` storage variable to `pointClaimed`.

Update

Clash of Coins' Response

We agree with the comment regarding naming; however, we do not plan to fix it as the contract is already released.

I-09	Missing error descriptions in `ClaimSoftCurrency`, `RetroDropWithMerkle`
Severity	**INFO**
Status	• FIXED

Location

File	Location	Line
ClaimSoftCurrency.sol	contract `ClaimSoftCurrency`	25
RetroDropWithMerkle.sol	contract `RetroDropWithMerkle`	18

Description

In the mentioned locations, there is no documentation explaining the errors and the scenarios in which they occur:

```
//ClaimSoftCurrency
error InvalidSigner();
error InvalidSignerAddress();

//RetroDropWithMerkle
error PointsAlreadyClaimed();
error InvalidProof();
error InvalidPoints();
```

Recommendation

We recommended adding documentation describing these errors and the cases in which they occur.

Update

Fixed at `0c8f42e93d6f7a6a6fdd84252c109150d3371ba4`

I-10	Unnecessary `<= 0` condition for `uint` in `ClaimRewards`
Severity	**INFO**
Status	• ACKNOWLEDGED

Location

File	Location	Line
ClaimRewards.sol	contract `ClaimRewards` > function `claimRewards`	70
ClaimRewards.sol	contract `ClaimRewards` > function `claimRewards`	74

Description

In the function `claimRewards` of the contract `ClaimRewards`, there are a few checks on `uint` type variables:

```
if (points <= 0) {
  revert InvalidPoints();
}

if (coins <= 0) {
  revert InvalidCoins();
}
```

However, using the condition `<= 0` is redundant for `uint` type variables, as they cannot be less than zero.

Recommendation

We recommend replacing `<= 0` with `== 0` for `uint` type variables.

Update

Clash of Coins' Response

We agree but cannot make changes as the contract has already been deployed to production.

3 DEPLOY VERIFICATION

OXORIO

3.1 SCOPE

Contract	Network	Address
`RetroDropWithMerkle`	Base	0x6d19e9bc21a2120ff6fe71aad28ecd9d05ed6973
`ClaimRewards`	Base	0x0fbBBd928EA4eDDd2EAfF51D4D412a3b65452F40

Deployment scripts:

`ClaimRewards` contract:

- ClaimRewards.ts

`RetroDropWithMerkle` contract:

- RetroDropWithMerkle.ts

Initialized roles:

`ClaimRewards`:

- Current `owner`: 0x4897a4eE9D0d4078cd5c39980A7b970Df392F590

`RetroDropWithMerkle`:

- Current `owner`: 0x4897a4eE9D0d4078cd5c39980A7b970Df392F590

3.2 CONCLUSION

The verification confirms the project’s security, with all checks receiving a "PASS" status. The contracts comply with specified standards, align with the architecture, and integrate auditor recommendations. Deployment scripts, bytecode, and initialization parameters are verified. Access control is well-structured, and documentation is comprehensive.

3.3.1 Network specific behavior

Status: PASS

All the network features affecting the protocol's operation are being studied. The virtual machine, the message transmission process within the main network, and vice versa (all distinctive network features and how they can impact the protocol's operation) are being researched.

Results

> Both contracts comply with compiler version `v0.8.27` and EVM version `Paris`.

3.3.2 Scope checking

Status: PASS

This stage involves auditors researching the provided scope for verification, studying project dependencies, and building the protocol's architecture. Project documentation is examined. Existing tests are also run at this stage, and the test coverage level is checked. Contract mocks are investigated for logical errors. The protocol's architecture is examined for conceptual errors.

Results

> The declared scope fully covers all dependent contracts and libraries and corresponds to the described architecture.

3.3.3 Audit report investigation

Status: PASS

At this stage, the presence of an audit report is verified, along with the alignment of the scope in the report with the deployed scope. It is checked whether all critical vulnerabilities have either been fixed or there is evidence that the vulnerability cannot be fixed without posing a threat to the protocol. Recommendations and the conclusion in the report are studied, as well as the alignment of the final commit with all the recommendations.

Results

> The contract code matches the audited commit and incorporates all the auditors' recommendations.

3.3.4 Deploy script check

Status: PASS

Auditors study the deployment script for contracts, examining initialization parameters. It is verified that interrupting the protocol deployment will not lead to incorrect initialization (for example, a front-run on initialization should result in both the script's reversion and require re-deployment).

Results

> In the deployment script `ignition/modules/ClaimRewards.ts`, the same owner address is used to set two constructor parameters: `backendSigner` and `owner`. However, this is not an issue, as the initialization parameters in the `ignition/parameters` folder are for demonstration purposes only.

3.3.5 Deployment verification

Status: PASS

The bytecode of the deployed contracts is checked to match the final commit in the report. An additional check is performed to verify all contracts on the explorer. Further verification is conducted to confirm that the bytecode of deployed contracts cannot be altered.

Results

> The bytecode of both compiled contracts matches the bytecode of the deployed contracts (except for the IPFS link at the end of the runtime bytecode).

3.3.6 Initialization parameters check

Status: PASS

At this stage, values are gathered from the storage in verified contracts, and they are checked for compliance with the parameters from the deployment script. Auditors ensure that all contracts are initialized and cannot be reinitialized by malicious users.

Results

> The initialization parameters align with the current storage values. It should be noted, however, that the initialization parameters in the `ignition/parameters` folder are for demonstration purposes only.

3.3.7 Role model verification

Status: PASS

The protocol's access control structure is examined to identify redundant roles or roles with more privileges than intended. It is checked that all access rights are set by the previously studied structure. If a role is assigned to a multisig, multisig owners are validated.

Results

> The `owner` role for the `ClaimRewards` contract was initially set to 0x9A8E0702bC77CB0456A776634A265A3efb19d44F during initialization, and ownership was subsequently transferred to the declared scope address 0x4897a4ee9d0d4078cd5c39980a7b970df392f590.
> In the second contract, `RetroDropWithMerkle`, the owner address 0x4897a4ee9d0d4078cd5c39980a7b970df392f590 specified in the scope is initially set during deployment.

3.3.8 Storage Check

Status: PASS

Contract storage check for the identification of potential vulnerabilities that could lead to unauthorized access to data or its modification.

Results

> All contract storage values are explicitly set during initialization, through authorized interactions with setters, or other contract functions.

3.3.9 Documentation Verification

Status: PASS

Document verification encompasses the analysis of functions and their passed values that directly modify the contract storage.

Results

> All functions and variables of both contracts are fully documented. The project also includes a comprehensive `README.md` file detailing all functionality.

4. APPENDIX

4.1 DISCLAIMER

At the request of client, Oxorio consents to the public release of this audit report. The information contained in this audit report is provided "as is," without any representations or warranties whatsoever. Oxorio disclaims any responsibility for damages that may arise from or in relation to this audit report. Oxorio retains copyright of this report.

The audit makes no statements or warranties about the utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, regulatory regime for the business model, or any other statements about the fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

4.2 SECURITY ASSESSMENT METHODOLOGY

Oxorio's smart contract audit methodology is designed to ensure the security, reliability, and compliance of smart contracts throughout their development lifecycle. Our process integrates the Smart Contract Security Verification Standard (SCSVS) with our advanced techniques to address complex security challenges. For a detailed look at our approach, please refer to the full version of our methodology. Here is a concise overview of our auditing process:

1. Project Architecture Review

All necessary information about the smart contract is gathered, including its intended functionality and dependencies. This stage sets the foundation by reviewing documentation, business logic, and initial code analysis.

2. Vulnerability Assessment

This phase involves a deep dive into the smart contract's code to identify security vulnerabilities. Rigorous testing and review processes are applied to ensure robustness against potential attacks.

This stage is focused on identifying specific vulnerabilities within the smart contract code. It involves scanning and testing the code for known security weaknesses and patterns that could potentially be exploited by malicious actors.

3. Security Model Evaluation

The smart contract’s architecture is assessed to ensure it aligns with security best practices and does not introduce potential vulnerabilities. This includes reviewing how the contract integrates with external systems, its compliance with security best practices, and whether the overall design supports a secure operational environment.

This phase involves a analysis of the project's documentation, the consistency of business logic as documented versus implemented in the code, and any assumptions made during the design and development phases. It assesses if the contract's architectural design adequately addresses potential threats and integrates necessary security controls.

4. Cross-Verification by Multiple Auditors

Typically, the project is assessed by multiple auditors to ensure a diverse range of insights and thorough coverage. Findings from individual auditors are cross-checked to verify accuracy and completeness.

5. Report Consolidation

Findings from all auditors are consolidated into a single, comprehensive audit report. This report outlines potential vulnerabilities, areas for improvement, and an overall assessment of the smart contract's security posture.

6. Reaudit of Revised Submissions

Post-review modifications made by the client are reassessed to ensure that all previously identified issues have been adequately addressed. This stage helps validate the effectiveness of the fixes applied.

7. Final Audit Report Publication

The final version of the audit report is delivered to the client and published on Oxorio's official website. This report includes detailed findings, recommendations for improvement, and an executive summary of the smart contract's security status.

4.3 CODEBASE QUALITY ASSESSMENT REFERENCE

The tables below describe the codebase quality assessment categories and rating criteria used in this report.

Category	Description
Access Control	Evaluates the effectiveness of mechanisms controlling access to ensure only authorized entities can execute specific actions, critical for maintaining system integrity and preventing unauthorized use.
Arithmetic	Focuses on the correct implementation of arithmetic operations to prevent vulnerabilities like overflows and underflows, ensuring that mathematical operations are both logically and semantically accurate.
Complexity	Assesses code organization and function clarity to confirm that functions and modules are organized for ease of understanding and maintenance, thereby reducing unnecessary complexity and enhancing readability.
Data Validation	Assesses the robustness of input validation to prevent common vulnerabilities like overflow, invalid addresses, and other malicious input exploits.
Decentralization	Reviews the implementation of decentralized governance structures to mitigate insider threats and ensure effective risk management during contract upgrades.
Documentation	Reviews the comprehensiveness and clarity of code documentation to ensure that it provides adequate guidance for understanding, maintaining, and securely operating the codebase.
External Dependencies	Evaluates the extent to which the codebase depends on external protocols, oracles, or services. It identifies risks posed by these dependencies, such as compromised data integrity, cascading failures, or reliance on centralized entities. The assessment checks if these external integrations have appropriate fallback mechanisms or redundancy to mitigate risks and protect the protocol's functionality.
Error Handling	Reviews the methods used to handle exceptions and errors, ensuring that failures are managed gracefully and securely.
Logging and Monitoring	Evaluates the use of event auditing and logging to ensure effective tracking of critical system interactions and detect potential anomalies.
Low-Level Calls	Reviews the use of low-level constructs like inline assembly, raw `call` or `delegatecall`, ensuring they are justified, carefully implemented, and do not compromise contract security.

Category	Description
Testing and Verification	Reviews the implementation of unit tests and integration tests to verify that codebase has comprehensive test coverage and reliable mechanisms to catch potential issues.

4.3.1 Rating Criteria

Rating	Description
Excellent	The system is flawless and surpasses standard industry best practices.
Good	Only minor issues were detected; overall, the system adheres to established best practices.
Fair	Issues were identified that could potentially compromise system integrity.
Poor	Numerous issues were identified that compromise system integrity.
Absent	A critical component is absent, severely compromising system safety.
Not Applicable	This category does not apply to the current evaluation.

4.4 FINDINGS CLASSIFICATION REFERENCE

4.4.1 Severity Level Reference

The following severity levels were assigned to the issues described in the report:

Title	Description
CRITICAL	Issues that pose immediate and significant risks, potentially leading to asset theft, inaccessible funds, unauthorized transactions, or other substantial financial losses. These vulnerabilities represent serious flaws that could be exploited to compromise or control the entire contract. They require immediate attention and remediation to secure the system and prevent further exploitation.
MAJOR	Issues that could cause a significant failure in the contract's functionality, potentially necessitating manual intervention to modify or replace the contract. These vulnerabilities may result in data corruption, malfunctioning logic, or prolonged downtime, requiring substantial operational changes to restore normal performance. While these issues do not immediately lead to financial losses, they compromise the reliability and security of the contract, demanding prioritized attention and remediation.
WARNING	Issues that might disrupt the contract's intended logic, affecting its correct functioning or making it vulnerable to Denial of Service (DDoS) attacks. These problems may result in the unintended triggering of conditions, edge cases, or interactions that could degrade the user experience or impede specific operations. While they do not pose immediate critical risks, they could impact contract reliability and require attention to prevent future vulnerabilities or disruptions.
INFO	Issues that do not impact the security of the project but are reported to the client's team for improvement. They include recommendations related to code quality, gas optimization, and other minor adjustments that could enhance the project's overall performance and maintainability.

4.4.2 Status Level Reference

Based on the feedback received from the client's team regarding the list of findings discovered by the contractor, the following statuses were assigned to the findings:

Title	Description
NEW	Waiting for the project team's feedback.

Title	Description
FIXED	Recommended fixes have been applied to the project code and the identified issue no longer affects the project's security.
ACKNOWLEDGED	The project team is aware of this finding and acknowledges the associated risks. This finding may affect the overall security of the project; however, based on the risk assessment, the team will decide whether to address it or leave it unchanged.
NO ISSUE	Finding does not affect the overall security of the project and does not violate the logic of its work.

4.5 ABOUT OXORIO

OXORIO is a blockchain security firm that specializes in smart contracts, zk-SNARK solutions, and security consulting. With a decade of blockchain development and five years in smart contract auditing, our expert team delivers premier security services for projects at any stage of maturity and development.

Since 2021, we've conducted key security audits for notable DeFi projects like Lido, 1Inch, Rarible, and deBridge, prioritizing excellence and long-term client relationships. Our co-founders, recognized by the Ethereum and Web3 Foundations, lead our continuous research to address new threats in the blockchain industry. Committed to the industry's trust and advancement, we contribute significantly to security standards and practices through our research and education work.

Our contacts:

- oxor.io
- ping@oxor.io
- Github
- Linkedin
- Twitter

THANK YOU FOR CHOOSING