OMNISCIA

December 28, 2024

SMART CONTRACT AUDIT REPORT

Perpetual Airdrop
Core AirDrop Implementation

omniscia.io

info@omniscia.io

Online report: perpetual-airdrop-core-airdrop-implementation

Omniscia.io is one of the fastest growing and most trusted blockchain security firms and has rapidly become a true market leader. To date, our team has collectively secured over 370+ clients, detecting 1,500+ high-severity issues in widely adopted smart contracts.

Founded in France at the start of 2020, and with a track record spanning back to 2017, our team has been at the forefront of auditing smart contracts, providing expert analysis and identifying potential vulnerabilities to ensure the highest level of security of popular smart contracts, as well as complex and sophisticated decentralized protocols.

Our clients, ecosystem partners, and backers include leading ecosystem players such as L'Oréal, Polygon, AvaLabs, Gnosis, Morpho, Vesta, Gravita, Olympus DAO, Fetch.ai, and LimitBreak, among others.

To keep up to date with all the latest news and announcements follow us on twitter @omniscia_sec.

omniscia.io

info@omniscia.io

Online report: perpetual-airdrop-core-airdrop-implementation

Core AirDrop Implementation Security Audit

Audit Report Revisions

Commit Hash	Date	Audit Report Hash
a5e8ff5ca3	December 20th 2024	f01ce762d0
61de1a0b9a	December 28th 2024	4d8a2b2b70
9fe8a4a3fc	December 28th 2024	7143ad38fb

Audit Overview

We were tasked with performing an audit of the Perpetual Airdrop codebase and in particular their Core AirDrop Implementation module.

The system represents a multi-token airdrop system that utilizes historical balances as well as on-transfer hooks to track eligibility.

Over the course of the audit, we identified that several misconfigurations are permitted albeit via the deployers of the system which are expected to apply proper sanitization to their configurations.

We advise the Perpetual Airdrop team to closely evaluate all minor-and-above findings identified in the report and promptly remediate them as well as consider all optimizational exhibits identified in the report.

Post-Audit Conclusion

The Perpetual Airdrop team iterated through all findings within the report and provided us with a revised commit hash to evaluate all exhibits on.

We evaluated all alleviations performed by Perpetual Airdrop and have identified that certain exhibits have not been adequately dealt with. We advise the Perpetual Airdrop team to revisit the following exhibits: `PAC-01M`, `PAC-03C`

Post-Audit Conclusion (9fe8a4a3fc)

The Perpetual Airdrop team provided us with a follow-up commit hash to evaluate the remediations of the two aforementioned exhibits.

We validated that both exhibits have been adequately addressed, and thus consider that all outputs of the audit report have been properly consumed by the Perpetual Airdrop team with no outstanding remediative actions remaining.

Audit Synopsis

Severity	Identified	Alleviated	Partially Alleviated	Acknowledged
Unknown	0	0	0	0
Informational	12	12	0	0
Minor	1	1	0	0
Medium	2	2	0	0
Major	0	0	0	0

During the audit, we filtered and validated a total of **6 findings utilizing static analysis** tools as well as identified a total of **9 findings during the manual review** of the codebase. We strongly recommend that any minor severity or higher findings are dealt with promptly prior to the project's launch as they can introduce potential misbehaviours of the system as well as exploits.

- 🎯 **Scope**
- 💻 **Compilation**
- 🔍 **Static Analysis**
- 👁️ **Manual Review**
- 🖊️ **Code Style**

Scope

The audit engagement encompassed a specific list of contracts that were present in the commit hash of the repository that was in scope. The tables below detail certain meta-data about the target of the security assessment and a navigation chart is present at the end that links to the relevant findings per file.

Target

- Repository: **https://github.com/perpetual-airdrop/contracts**
- Commit: a5e8ff5ca31fe8c5fc57732866142ce01ab9a49c
- Language: Solidity
- Network: Ethereum
- Revisions: **a5e8ff5ca3**, **61de1a0b9a**, **9fe8a4a3fc**

Contracts Assessed

File	Total Finding(s)
contracts/AirdropSourceToken.sol (AST)	1
contracts/DatetimeLibrary.sol (DLY)	2
contracts/PerpetualAirdropToken.sol (PAT)	3
contracts/types/PerpetualAirdropTypes.sol (PAS)	0
contracts/PerpetualAirdropCoordinator.sol (PAC)	7
contracts/TripleAirdrop.sol (TAP)	2

Compilation

The project utilizes `hardhat` as its development pipeline tool, containing an array of tests and scripts coded in TypeScript.

To compile the project, the `compile` command needs to be issued via the `npx` CLI tool to `hardhat`:

```bash
npx hardhat compile
```

The `hardhat` tool automatically selects Solidity version `0.8.20` based on the version specified within the `hardhat.config.ts` file.

The project contains discrepancies with regards to the Solidity version used as the `pragma` statements of the contracts are open-ended (`^0.8.20`).

We advise them to be locked to `0.8.20` (`=0.8.20`), the same version utilized for our static analysis as well as optimizational review of the codebase.

During compilation with the `hardhat` pipeline, no errors were identified that relate to the syntax or bytecode size of the contracts.

Static Analysis

The execution of our static analysis toolkit identified **47 potential issues** within the codebase of which **36 were ruled out to be false positives** or negligible findings.

The remaining **11 issues** were validated and grouped and formalized into the **6 exhibits** that follow:

ID	Severity	Addressed	Title
DLY-01S	Informational	Yes	Illegible Numeric Value Representations
PAC-01S	Informational	Yes	Inexistent Event Emission
PAC-02S	Informational	Yes	Inexistent Sanitization of Input Address
PAC-03S	Informational	Yes	Inexistent Visibility Specifier
PAT-01S	Informational	Yes	Inexistent Event Emissions
PAT-02S	Informational	Yes	Inexistent Sanitization of Input Addresses

Manual Review

A **thorough line-by-line review** was conducted on the codebase to identify potential malfunctions and vulnerabilities in Perpetual Airdrop.

As the project at hand implements Perpetual Airdrop, intricate care was put into ensuring that the **flow of funds within the system conforms to the specifications and restrictions** laid forth within the protocol's specification.

We validated that **all state transitions of the system occur within sane criteria** and that all rudimentary formulas within the system execute as expected. We **pinpointed multiple potential misconfigurations permitted** within the system which could have had **minor-to-moderate ramifications** to its overall operation; we urge the Perpetual Airdrop team to closely evaluate all minor-and-above exhibits within the audit report.

Additionally, the system was investigated for any other commonly present attack vectors such as re-entrancy attacks, mathematical truncations, logical flaws and **ERC / EIP** standard inconsistencies. The documentation of the project was satisfactory to the extent it need be.

A total of **9 findings** were identified over the course of the manual review of which **3 findings** concerned the behaviour and security of the system. The non-security related findings, such as optimizations, are included in the separate **Code Style** chapter.

The finding table below enumerates all these security / behavioural findings:

ID	Severity	Addressed	Title
DLY-01M	Medium	Yes	Inexistent Subtraction of Year
PAC-01M	Medium	Yes	Inexistent Validation of Non-Zero Winners
TAP-01M	Minor	Yes	Inexistent Prevention of Duplicate Token Entries

Code Style

During the manual portion of the audit, we identified **6 optimizations** that can be applied to the codebase that will decrease the operational cost associated with the execution of a particular function and generally ensure that the project complies with the latest best practices and standards in Solidity.

Additionally, this section of the audit contains any opinionated adjustments we believe the code should make to make it more legible as well as truer to its purpose.

These optimizations are enumerated below:

ID	Severity	Addressed	Title
AST-01C	Informational	Yes	Inefficient Re-Reservation of Memory
PAC-01C	Informational	Yes	Ineffectual Usage of Safe Arithmetics
PAC-02C	Informational	Yes	Inefficient Indices Initialization
PAC-03C	Informational	Yes	Inefficient `mapping` Lookups
PAT-01C	Informational	Yes	Improper Use-Case Permittance
TAP-01C	Informational	Yes	Redundant Multi-Entry Eligibility Mechanism

DatetimeLibrary Static Analysis Findings

DLY-01S: Illegible Numeric Value Representations

Type	Severity	Location
Code Style	Informational	**DatetimeLibrary.sol**: • I-1: **L5** • I-2: **L6**

Description:

The linked representations of numeric literals are sub-optimally represented decreasing the legibility of the codebase.

Example:

contracts/DatetimeLibrary.sol

SOL

```
uint256 constant OFFSET19700101 = 2440588;
```

Recommendation:

To properly illustrate each value's purpose, we advise the following guidelines to be followed. For values meant to depict fractions with a base of `1e18`, we advise fractions to be utilized directly (i.e. `1e17` becomes `0.1e18`) as they are supported. For values meant to represent a percentage base, we advise each value to utilize the underscore (`_`) separator to discern the percentage decimal (i.e. `10000` becomes `100_00`, `300` becomes `3_00` and so on). Finally, for large numeric values we simply advise the underscore character to be utilized again to represent them (i.e. `1000000` becomes `1_000_000`).

Alleviation (61de1a0b9a46e240dd85f368896ad9409af7358f):

The number literals were appropriately adjusted, introducing the underscore character in the `OFFSET19700101` declaration whilst using a more explicit multiplication for the seconds in a day.

PerpetualAirdropCoordinator Static Analysis Findings

PAC-01S: Inexistent Event Emission

Type	Severity	Location
Language Specific	Informational	PerpetualAirdropCoordinator.sol:L142-L144

Description:

The linked function adjusts a sensitive contract variable yet does not emit an event for it.

Example:

contracts/PerpetualAirdropCoordinator.sol

SOL

```
142 function setRandomnessProvider(address _randomnessProviderAddress) external onlyOwner {
143     randomnessProvider = IRandomnessProvider(_randomnessProviderAddress);
144 }
```

Recommendation:

We advise an `event` to be declared and correspondingly emitted to ensure off-chain processes can properly react to this system adjustment.

Alleviation (61de1a0b9a46e240dd85f368896ad9409af7358f):

The `RandomnessProviderSet` event was introduced to the codebase and is correspondingly emitted in the **`PerpetualAirdropCoordinator::setRandomnessProvider`** function, addressing this exhibit in full.

PAC-02S: Inexistent Sanitization of Input Address

Type	Severity	Location
Input Sanitization	Informational	PerpetualAirdropCoordinator.sol:L142-L144

Description:

The linked function accepts an `address` argument yet does not properly sanitize it.

Impact:

The presence of zero-value addresses, especially in `constructor` implementations, can cause the contract to be permanently inoperable. These checks are advised as zero-value inputs are a common side-effect of off-chain software related bugs.

Example:

contracts/PerpetualAirdropCoordinator.sol

SOL

```
142 function setRandomnessProvider(address _randomnessProviderAddress) external onlyOwner {
143     randomnessProvider = IRandomnessProvider(_randomnessProviderAddress);
144 }
```

Recommendation:

We advise some basic sanitization to be put in place by ensuring that the `address` specified is non-zero.

Alleviation (61de1a0b9a46e240dd85f368896ad9409af7358f):

The input `_randomnessProviderAddress` address argument of the **`PerpetualAirdropCoordinator::setRandomnessProvider`** function is adequately sanitized as non-zero in the latest in-scope revision of the codebase, addressing this exhibit.

PAC-03S: Inexistent Visibility Specifier

Type	Severity	Location
Code Style	Informational	PerpetualAirdropCoordinator.sol:L28

Description:

The linked variable has no visibility specifier explicitly set.

Example:

contracts/PerpetualAirdropCoordinator.sol

SOL

```sol
28 uint32 numRegularAirdropWinners;
```

Recommendation:

We advise one to be set so to avoid potential compilation discrepancies in the future as the current behaviour is for the compiler to assign one automatically which may deviate between `pragma` versions.

Alleviation (61de1a0b9a46e240dd85f368896ad9409af7358f):

The `public` visibility specifier has been introduced to the referenced variable, preventing potential compilation discrepancies and addressing this exhibit.

PerpetualAirdropToken Static Analysis Findings

PAT-01S: Inexistent Event Emissions

Type	Severity	Location
Language Specific	Informational	**PerpetualAirdropToken.sol**: • I-1: **L34-L36** • I-2: **L38-L40** • I-3: **L42-L44**

Description:

The linked functions adjust sensitive contract variables yet do not emit an event for it.

Example:

contracts/PerpetualAirdropToken.sol

SOL

```
function setCoordinator(address _coordinator) external onlyOwner {
    coordinator = _coordinator;
}
```

Recommendation:

We advise an `event` to be declared and correspondingly emitted for each function to ensure off-chain processes can properly react to this system adjustment.

Alleviation (61de1a0b9a46e240dd85f368896ad9409af7358f):

The `CoordinatorSet`, `TransactionLoggerAdded`, and `TransactionLoggerRemoved` events were introduced to the codebase and are correspondingly emitted in the **`PerpetualAirdropToken::setCoordinator`**, **`PerpetualAirdropToken::addTransactionLogger`**, and **`PerpetualAirdropToken::removeTransactionLogger`** functions respectively, addressing this exhibit in full.

PAT-02S: Inexistent Sanitization of Input Addresses

Type	Severity	Location
Input Sanitization	Informational	**PerpetualAirdropToken.sol**: • I-1: **L34-L36** • I-2: **L38-L40** • I-3: **L42-L44**

Description:

The linked function(s) accept `address` arguments yet do not properly sanitize them.

Impact:

The presence of zero-value addresses, especially in `constructor` implementations, can cause the contract to be permanently inoperable. These checks are advised as zero-value inputs are a common side-effect of off-chain software related bugs.

Example:

contracts/PerpetualAirdropToken.sol

SOL

```
function setCoordinator(address _coordinator) external onlyOwner {
    coordinator = _coordinator;
}
```

Recommendation:

We advise some basic sanitization to be put in place by ensuring that each `address` specified is non-zero.

Alleviation (61de1a0b9a46e240dd85f368896ad9409af7358f):

All input argument(s) of the `PerpetualAirdropToken::setCoordinator`, `PerpetualAirdropToken::addTransactionLogger`, and `PerpetualAirdropToken::removeTransactionLogger` functions are adequately sanitized as non-zero in the latest in-scope revision of the codebase, addressing this exhibit.

DatetimeLibrary Manual Review Findings

DLY-01M: Inexistent Subtraction of Year

Type	Severity	Location
Logical Fault	Medium	DatetimeLibrary.sol:L24

Description:

The `DatetimeLibrary::_daysToMonth` function does not properly subtract the year from the `_month` calculation to evaluate the actual month date.

Impact:

The `DatetimeLibrary::_daysToMonth` function will yield continuously inflated values with the years included.

Example:

contracts/DatetimeLibrary.sol

SOL

```
20 function _daysToMonth(uint256 _days) internal pure returns (uint256 month) {
21     uint256 L = _days + 68569 + OFFSET19700101;
22     uint256 N = 4 * L / 146097;
23     L = L - (146097 * N + 3) / 4;
24     L = L + 31;
25     uint256 _month = 80 * L / 2447;
26     L = _month / 11;
27     _month = _month + 2 - 12 * L;
28 
29     month = _month;
```

Example (Cont.):

```sol
}
```

Recommendation:

We advise the year to be removed per the **original implementation** based on the JD formula.

Alleviation (61de1a0b9a46e240dd85f368896ad9409af7358f):

The code of the `DatetimeLibrary::_daysToMonth` function (now renamed to `DatetimeLibrary::_daysToDate`) was updated to reflect the original implementation ensuring that the month is appropriately evaluated.

PerpetualAirdropCoordinator Manual Review Findings

PAC-01M: Inexistent Validation of Non-Zero Winners

Type	Severity	Location
Input Sanitization	Medium	PerpetualAirdropCoordinator.sol:L117

Description:

The `PerpetualAirdropCoordinator::_setRegularAirdropConfig` function does not validate that a non-zero amount of airdrop winners have been defined.

Impact:

The reward distribution mechanism will fail to execute properly if a distribution has been defined with zero winners as no distributions beyond it will be processed.

Example:

contracts/PerpetualAirdropCoordinator.sol

SOL

```
114 // Set regular distributions
115 for (uint256 i = 0; i < _config.distributions.length; i++) {
116     RegularDistribution memory _distribution = _config.distributions[i];
117     numRegularAirdropWinners += _distribution.numWinners;
118
119     // Create a new RegularDistribution in storage
120     regularAirdropConfig.distributions.push();
121     RegularDistribution storage newDistribution = regularAirdropConfig.distributions[
122         regularAirdropConfig.distributions.length - 1
123     ];
```

Example (Cont.):

```sol
    // Set the basic properties
    newDistribution.sourceToken = _distribution.sourceToken;
    newDistribution.numWinners = _distribution.numWinners;
    newDistribution.distributionType = _distribution.distributionType;

    // Copy the distributions array
    for (uint256 j = 0; j < _distribution.distributions.length; j++) {
        newDistribution.distributions.push(
            Distribution({
                token: _distribution.distributions[j].token,
                amount: _distribution.distributions[j].amount
            })
        );
    }
}
```

Recommendation:

We advise such restrictions to be imposed, ensuring that no misbehaviours may arise when distributing rewards from a regular airdrop.

Alleviation (61de1a0b9a):

While a non-zero check has been introduced for the `numRegularAirdropWinners` variable, no such check was introduced for the actual `_distribution.numWinners` variable which renders this exhibit partially addressed.

Alleviation (9fe8a4a3fc):

The code was updated further to ensure that the `_distribution.numWinners` is non-zero for each entry, alleviating this exhibit in full.

TripleAirdrop Manual Review Findings

TAP-01M: Inexistent Prevention of Duplicate Token Entries

Type	Severity	Location
Input Sanitization	Minor	TripleAirdrop.sol:L23-L25

Description:

Any duplicate token entry in the `_tokens` configured for a `TripleAirdrop` will result in double accounting of balances and should be disallowed.

Impact:

A `TripleAirdrop` defined with duplicate entries will misbehave in its cumulative accounting.

Example:

contracts/TripleAirdrop.sol

SOL

```
23 for (uint256 i = 0; i < _tokens.length; i++) {
24     tokens.add(_tokens[i]);
25 }
```

Recommendation:

We advise the `TripleAirdrop::constructor` to prevent duplicate entries either by requiring that the value yielded by the `EnumerableSet::add` function is `true` or by mandating that the tokens are defined in a strictly ascending order, either of which we consider an adequate resolution to this exhibit.

Alleviation (61de1a0b9a46e240dd85f368896ad9409af7358f):

A `require` check was introduced as advised, ensuring duplicate tokens cannot be introduced.

AirdropSourceToken Code Style Findings

AST-01C: Inefficient Re-Reservation of Memory

Type	Severity	Location
Gas Optimization	Informational	AirdropSourceToken.sol:L22, L34

Description:

The referenced `return` statement can yield the already-reserved `winners` variable directly.

Example:

contracts/AirdropSourceToken.sol

SOL

```
22 address[] memory winners = new address[](numWinners);
23 uint256 totalLikelihood;
24 uint256[] memory cumListLikelihoods = new uint256[](numLists);
25
26 // Calculate the likelihood for each list
27 for (uint8 i = 0; i < numLists; i++) {
28     uint256 listLikelihood = balanceThresholds[i] * eligibilityLists[i].length();
29     totalLikelihood += listLikelihood;
30     cumListLikelihoods[i] = totalLikelihood;
31 }
```

Example (Cont.):

```sol

if (totalLikelihood == 0) {
    return new address[](numWinners);
}
```

Recommendation:

We advise this to be done so, optimizing the code's gas cost.

Alleviation (61de1a0b9a46e240dd85f368896ad9409af7358f):

The code was optimized as advised, yielding the already-declared `winners` array in an optimal way.

PerpetualAirdropCoordinator Code Style Findings

PAC-01C: Ineffectual Usage of Safe Arithmetics

Type	Severity	Location
Language Specific	Informational	PerpetualAirdropCoordinator.sol:L302

Description:

The linked mathematical operation is guaranteed to be performed safely by logical inference, such as surrounding conditionals evaluated in `require` checks or `if-else` constructs.

Example:

contracts/PerpetualAirdropCoordinator.sol

SOL

```
lastIndex--;
```

Recommendation:

Given that safe arithmetics are toggled on by default in `pragma` versions of `0.8.X`, we advise the linked statement to be wrapped in an `unchecked` code block thereby optimizing its execution cost.

Alleviation (61de1a0b9a46e240dd85f368896ad9409af7358f):

The referenced operation has been safely wrapped in an `unchecked` code block optimizing its gas cost.

PAC-02C: Inefficient Indices Initialization

Type	Severity	Location
Gas Optimization	Informational	PerpetualAirdropCoordinator.sol:L281-L283

Description:

The referenced indices are inefficiently initialized as the code could simply utilize the participants directly.

Example:

contracts/PerpetualAirdropCoordinator.sol

SOL

```
276 // Assign extra amounts using randomWords
277 if (remainder > 0) {
278     uint256[] memory available = new uint256[](numParticipants);
279 
280     // Initialize available indices
281     for (uint256 i = 0; i < numParticipants; i++) {
282         available[i] = i;
283     }
284 
285     uint256 lastIndex = numParticipants;
```

Example (Cont.):

SOL

```
    for (uint256 i = 0; i < remainder; i++) {
        uint256 randomIndex = randomWords[i] % lastIndex;
        uint256 selectedIndex = available[randomIndex];
        address selectedParticipant = participants[selectedIndex];

        for (uint256 j = 0; j < numDistributions; j++) {
            Distribution memory distribution = initialAirdropDistributions[j];
            EnumerableMap.AddressToUintMap storage tokenEarnings = earnings[
                distribution.token
            ];

            (, uint256 currentAmount) = tokenEarnings.tryGet(selectedParticipant);
            tokenEarnings.set(selectedParticipant, currentAmount + distribution.amount);
        }

        // Move the picked participant to the end of the array to avoid re-selection
        lastIndex--;
        if (randomIndex != lastIndex) {
            available[randomIndex] = available[lastIndex];
        }
    }
}
```

Recommendation:

We advise the participants to be utilized directly, optimizing the code's gas cost significantly.

Alleviation (61de1a0b9a46e240dd85f368896ad9409af7358f):

The code was updated per our recommendation, optimizing its gas cost significantly.

PAC-03C: Inefficient `mapping` Lookups

Type	Severity	Location
Gas Optimization	Informational	PerpetualAirdropCoordinator.sol:L531, L536

Description:

The linked statements perform key-based lookup operations on `mapping` declarations from storage multiple times for the same key redundantly.

Example:

contracts/PerpetualAirdropCoordinator.sol

SOL

```
require(earnings[token].contains(account), 'No earnings');

uint256 amount = earnings[token].get(account);

// Update state
earnings[token].remove(account);
```

Recommendation:

As the lookups internally perform an expensive `keccak256` operation, we advise the lookups to be cached wherever possible to a single local declaration that either holds the value of the `mapping` in case of primitive types or holds a `storage` pointer to the `struct` contained.

As the compiler's optimizations may take care of these caching operations automatically at-times, we advise the optimization to be selectively applied, tested, and then fully adopted to ensure that the proposed caching model indeed leads to a reduction in gas costs.

Alleviation (61de1a0b9a):

While the exhibit was marked as addressed in the GitHub repository, no alleviation was observed for it rendering it to remain open.

Alleviation (9fe8a4a3fc):

The referenced `mapping` lookup pair has been optimized by storing the `earnings[token]` lookup to a local `storage` variable, optimizing the codebase as advised.

PerpetualAirdropToken Code Style Findings

PAT-01C: Improper Use-Case Permittance

Type	Severity	Location
Standard Conformity	Informational	PerpetualAirdropToken.sol:L131-L144

Description:

The `PerpetualAirdropToken::delegateBySig` function will only permit a signed payload by the caller itself which renders it redundant.

Example:

contracts/PerpetualAirdropToken.sol

SOL

```
/**
 * @dev Restrict delegation to only allow self-delegation.
 */
function delegate(address delegatee) public override {
    require(
        delegatee == msg.sender,
        'Can only delegate to yourself'
    );
    super.delegate(delegatee);
}
```

Example (Cont.):

```sol

/**
 * @dev Restrict delegation by signature to only allow self-delegation.
 */
function delegateBySig(
    address delegatee,
    uint256 nonce,
    uint256 expiry,
    uint8 v,
    bytes32 r,
    bytes32 s
) public override {
    require(
        delegatee == msg.sender,
        'RestrictedDelegationToken: Can only delegate to yourself'
    );
    super.delegateBySig(delegatee, nonce, expiry, v, r, s);
}
```

Recommendation:

We advise it to be restricted altogether, ensuring callers invoke the `PerpetualAirdropToken::delegate` function instead.

Alleviation (61de1a0b9a46e240dd85f368896ad9409af7358f):

Signature-based delegation is properly restricted in the contract as advised, addressing this inconsistency.

TripleAirdrop Code Style Findings

TAP-01C: Redundant Multi-Entry Eligibility Mechanism

Type	Severity	Location
Gas Optimization	Informational	TripleAirdrop.sol:L29, L70-L81

Description:

The `TripleAirdrop` will maintain multiple eligibility lists yet the balance threshold for each will be the same, meaning that maintaining each one is incorrect.

Example:

contracts/TripleAirdrop.sol

SOL

```
27 for (uint256 i = 0; i < numLists; i++) {
28     eligibilityLists.push();
29     balanceThresholds.push(_tokens.length * _singleTokenBalanceThreshold);
30 }
```

Recommendation:

We advise the system to maintain a single eligibility list, greatly optimizing its gas cost.

Alleviation (61de1a0b9a46e240dd85f368896ad9409af7358f):

The multi-entry eligibility mechanism was updated to incorporate the iterator in the calculation of the threshold, ensuring that there is meaning in the distinct balance thresholds and thus addressing this exhibit.

Finding Types

A description of each finding type included in the report can be found below and is linked by each respective finding. A full list of finding types Omniscia has defined will be viewable at the central audit methodology we will publish soon.

Input Sanitization

As there are no inherent guarantees to the inputs a function accepts, a set of guards should always be in place to sanitize the values passed in to a particular function.

Indeterminate Code

These types of issues arise when a linked code segment may not behave as expected, either due to mistyped code, convoluted if blocks, overlapping functions / variable names and other ambiguous statements.

Language Specific

Language specific issues arise from certain peculiarities that the Circom language boasts that discerns it from other conventional programming languages.

Curve Specific

Circom defaults to using the BN128 scalar field (a 254-bit prime field), but it also supports BSL12-381 (which has a 255-bit scalar field) and Goldilocks (with a 64-bit scalar field). However, since there are no constants denoting either the prime or the prime size in bits available in the Circom language, some Circomlib templates like `Sign` (which returns the sign of the input signal), and `AliasCheck` (used by the strict versions of `Num2Bits` and `Bits2Num`), hardcode either the BN128 prime size or some other constant related to BN128. Using these circuits with a custom prime may thus lead to unexpected results and should be avoided.

Code Style

In these types of findings, we identify whether a project conforms to a particular naming convention and whether that convention is consistent within the codebase and legible. In case of inconsistencies, we point them out under this category. Additionally, variable shadowing falls under this category as well which is identified when a local-level variable contains the same name as a toplevel variable in the circuit.

Mathematical Operations

This category is used when a mathematical issue is identified. This implies an issue with the implementation of a calculation compared to the specifications.

Logical Fault

This category is a bit broad and is meant to cover implementations that contain flaws in the way they are implemented, either due to unimplemented functionality, unaccounted-for edge cases or similar extraordinary scenarios.

Privacy Concern

This category is used when information that is meant to be kept private is made public in some way.

Proof Concern

Under-constrained signals are one of the most common issues in zero-knowledge circuits. Issues with proof generation fall under this category.

Severity Definition

In the ever-evolving world of blockchain technology, vulnerabilities continue to take on new forms and arise as more innovative projects manifest, new blockchain-level features are introduced, and novel layer-2 solutions are launched. When performing security reviews, we are tasked with classifying the various types of vulnerabilities we identify into subcategories to better aid our readers in understanding their impact.

Within this page, we will clarify what each severity level stands for and our approach in categorizing the findings we pinpoint in our audits. To note, all severity assessments are performed **as if the contract's logic cannot be upgraded** regardless of the underlying implementation.

Severity Levels

There are five distinct severity levels within our reports; `unknown`, `informational`, `minor`, `medium`, and `major`. A TL;DR overview table can be found below as well as a dedicated chapter to each severity level:

	Impact (None)	Impact (Low)	Impact (Moderate)	Impact (High)
Likelihood (None)	Informational	Informational	Informational	Informational
Likelihood (Low)	Informational	Minor	Minor	Medium
Likelihood (Moderate)	Informational	Minor	Medium	Major
Likelihood (High)	Informational	Medium	Major	Major

Unknown Severity

The `unknown` severity level is reserved for misbehaviors we observe in the codebase that cannot be quantified using the above metrics. Examples of such vulnerabilities include potentially desirable system behavior that is undocumented, reliance on external dependencies that are out-of-scope but could result in some form of vulnerability arising, use of external out-of-scope contracts that appears incorrect but cannot be pinpointed, and other such vulnerabilities.

In general, `unknown` severity level vulnerabilities require follow-up information by the project being audited and are either adjusted in severity (if valid), or marked as nullified (if invalid).

Additionally, the `unknown` severity level is sometimes assigned to centralization issues that cannot be assessed in likelihood due to their exploitation being tied to the honesty of the project's team.

Informational Severity

The `informational` severity level is dedicated to findings that do not affect the code functionally and tend to be stylistic or optimizational in nature. Certain edge cases are also set under `informational` vulnerabilities, such as overflow operations that will not manifest in the lifetime of the contract but should be guarded against as a best practice, to give an example.

Minor Severity

The `minor` severity level is meant for vulnerabilities that require functional changes in the code but tend to either have little impact or be unlikely to be recreated in a production environment. These findings can be acknowledged except for findings with a moderate impact but low likelihood which must be alleviated.

Medium Severity

The `medium` severity level is assigned to vulnerabilities that must be alleviated and have an observable impact on the overall project. These findings can only be acknowdged if the project deems them desirable behavior and we disagree with their point-of-view, instead urging them to reconsider their stance while marking the exhibit as acknowledged given that the project has ultimate say as to what vulnerabilities they end up patching in their system.

Major Severity

The `major` severity level is the maximum that can be specified for a finding and indicates a significant flaw in the code that must be alleviated.

Likelihood & Impact Assessment

As the preface chapter specifies, the blockchain space is constantly reinventing itself meaning that new vulnerabilities take place and our understanding of what security means differs year-to-year.

In order to reliably assess the likelihood and impact of a particular vulnerability, we instead apply an abstract measurement of a vulnerability's impact, duration the impact is applied for, and probability that the vulnerability would be exploited in a production environment.

Our proposed definitions are inspired by multiple sources in the security community and are as follows:

- Impact (High): A core invariant of the protocol can be broken for an extended duration.
- Impact (Moderate): A non-core invariant of the protocol can be broken for an extended duration or at scale, or an otherwise major-severity issue is reduced due to hypotheticals or external factors affecting likelihood.
- Impact (Low): A non-core invariant of the protocol can be broken with reduced likelihood or impact.
- Impact (None): A code or documentation flaw whose impact does not achieve low severity, or an issue without theoretical impact; a valuable best-practice
- Likelihood (High): A flaw in the code that can be exploited trivially and is ever-present.
- Likelihood (Moderate): A flaw in the code that requires some external factors to be exploited that are likely to manifest in practice.
- Likelihood (Low): A flaw in the code that requires multiple external factors to be exploited that may manifest in practice but would be unlikely to do so.
- Likelihood (None): A flaw in the code that requires external factors proven to be impossible in a production environment, either due to mathematical constraints, operational constraints, or system-related factors (i.e. EIP-20 tokens not being re-entrant).

Disclaimer

The following disclaimer applies to all versions of the audit report produced (preliminary / public / private) and is in effect for all past, current, and future audit reports that are produced and hosted under Omniscia:

IMPORTANT TERMS & CONDITIONS REGARDING OUR SECURITY AUDITS/REVIEWS/REPORTS AND ALL PUBLIC/PRIVATE CONTENT/DELIVERABLES

Omniscia ("Omniscia") has conducted an independent security review to verify the integrity of and highlight any vulnerabilities, bugs or errors, intentional or unintentional, that may be present in the codebase that were provided for the scope of this Engagement.

Blockchain technology and the cryptographic assets it supports are nascent technologies. This makes them extremely volatile assets. Any assessment report obtained on such volatile and nascent assets may include unpredictable results which may lead to positive or negative outcomes.

In some cases, services provided may be reliant on a variety of third parties. This security review does not constitute endorsement, agreement or acceptance for the Project and technology that was reviewed. Users relying on this security review should not consider this as having any merit for financial advice or technological due diligence in any shape, form or nature.

The veracity and accuracy of the findings presented in this report relate solely to the proficiency, competence, aptitude and discretion of our auditors. Omniscia and its employees make no guarantees, nor assurance that the contracts are free of exploits, bugs, vulnerabilities, deprecation of technologies or any system / economical / mathematical malfunction.

This audit report shall not be printed, saved, disclosed nor transmitted to any persons or parties on any objective, goal or justification without due written assent, acquiescence or approval by Omniscia.

All the information/opinions/suggestions provided in this report does not constitute financial or investment advice, nor should it be used to signal that any person reading this report should invest their funds without sufficient individual due diligence regardless of the findings presented in this report.

Information in this report is provided 'as is'. Omniscia is under no covenant to the completeness, accuracy or solidity of the contracts reviewed. Omniscia's goal is to help reduce the attack vectors/surface and the high level of variance associated with utilizing new and consistently changing technologies.

Omniscia in no way claims any guarantee, warranty or assurance of security or functionality of the technology that was in scope for this security review.

In no event will Omniscia, its partners, employees, agents or any parties related to the design/creation of this security review be ever liable to any parties for, or lack thereof, decisions and/or actions with regards to the information provided in this security review.

Cryptocurrencies and all other technologies directly or indirectly related to cryptocurrencies are not standardized, highly prone to malfunction and extremely speculative by nature. No due diligence and/or safeguards may be insufficient and users should exercise maximum caution when participating and/or investing in this nascent industry.

The preparation of this security review has made all reasonable attempts to provide clear and actionable recommendations to the Project team (the “client”) with respect to the rectification, amendment and/or revision of any highlighted issues, vulnerabilities or exploits within the contracts in scope for this engagement.

It is the sole responsibility of the Project team to provide adequate levels of test and perform the necessary checks to ensure that the contracts are functioning as intended, and more specifically to ensure that the functions contained within the contracts in scope have the desired intended effects, functionalities and outcomes, as documented by the Project team.

All services, the security reports, discussions, work product, attack vectors description or any other materials, products or results of this security review engagement is provided "as is" and "as available" and with all faults, uncertainty and defects without warranty or guarantee of any kind.

Omniscia will assume no liability or responsibility for delays, errors, mistakes, or any inaccuracies of content, suggestions, materials or for any loss, delay, damage of any kind which arose as a result of this engagement/security review.

Omniscia will assume no liability or responsibility for any personal injury, property damage, of any kind whatsoever that resulted in this engagement and the customer having access to or use of the products, engineers, services, security report, or any other other materials.

For avoidance of doubt, this report, its content, access, and/or usage thereof, including any associated services or materials, shall not be considered or relied upon as any form of financial, investment, tax, legal, regulatory, or any other type of advice.