OMNISCIA

April 2, 2025

SMART CONTRACT AUDIT REPORT

Perpetual Airdrop Project
Migration Compatibility Update

omniscia.io

info@omniscia.io

Online report: perpetual-airdrop-project-migration-compatibility-upd

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-project-migration-compatibility-update

Migration Compatibility Update Security Audit

Audit Report Revisions

Commit Hash	Date	Audit Report Hash
5d77092b77	February 20th 2025	6f2a34ac7d
bdef412a37	February 24th 2025	9f28496f1a
bef318aa99	February 26th 2025	42f9a4d4f0
bef318aa99	April 2nd 2025	963d171a3e

Audit Overview

We were tasked with performing an audit of the Perpetual Airdrop Project codebase and in particular their migration-compatible system upgrade.

Over the course of the audit, we identified several optimizations as well as a potential issue in the `Token0ne` implementation in relation to total supply tracking of governance balances.

We advise the Perpetual Airdrop Project 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 Project 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 Project and have identified that a particular exhibit has not been adequately dealt with. We advise the Perpetual Airdrop Project team to revisit the following exhibit: `TOE-02M`

Post-Audit Conclusion (bef318aa99)

The Perpetual Airdrop Project team evaluated our follow-up recommendation on the aforementioned exhibit and proceeded to remediate it in full.

We consider all outputs of the audit report properly consumed by the Perpetual Airdrop Project team with no remediative actions remaining.

Audit Synopsis

Severity	Identified	Alleviated	Partially Alleviated	Acknowledged
Unknown	0	0	0	0
Informational	7	7	0	0
Minor	0	0	0	0
Medium	3	3	0	0
Major	0	0	0	0

During the audit, we filtered and validated a total of **3 findings utilizing static analysis** tools as well as identified a total of **7 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: 5d77092b774f6680a62903c5ede6d10abb0b9f9a
- Language: Solidity
- Network: EVM
- Revisions: 5d77092b77, bdef412a37, bef318aa99

Contracts Assessed

File	Total Finding(s)
contracts/Migration.sol (MNO)	1
contracts/PerpetualAirdropToken.sol (PAT)	3
contracts/PerpetualAirdropCoordinator.sol (PAC)	2
contracts/TokenOne.sol (TOE)	4

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 **19 potential issues** within the codebase of which **16 were ruled out to be false positives** or negligible findings.

The remaining **3 issues** were validated and grouped and formalized into the **3 exhibits** that follow:

ID	Severity	Addressed	Title
MNO-01S	Medium	Yes	Improper Invocation of EIP-20 `transferFrom`
PAT-01S	Informational	Yes	Inexistent Sanitization of Input Address
TOE-01S	Informational	Yes	Inexistent Sanitization of Input Address

Manual Review

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

As the project at hand implements migration-compatible contracts, 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 a single vulnerability** within the system which could have had **moderate ramifications** to its overall operation; for more information, kindly consult the relevant medium-severity exhibit 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 **7 findings** were identified over the course of the manual review of which **2 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
TOE-01M	Medium	Yes	Incorrect Voter Threshold Imposition
TOE-02M	Medium	Yes	Inexistent Update of Fulfilment Status

Code Style

During the manual portion of the audit, we identified **5 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
PAC-01C	Informational	Yes	Inefficient Loop Limit Evaluation
PAC-02C	Informational	Yes	Inefficient `mapping` Lookups
PAT-01C	Informational	Yes	Inefficient Iterator Type
PAT-02C	Informational	Yes	Inefficient Loop Limit Evaluation
TOE-01C	Informational	Yes	Inefficient Calculations

Migration Static Analysis Findings

MNO-01S: Improper Invocation of EIP-20 `transferFrom`

Type	Severity	Location
Standard Conformity	Medium	Migration.sol:L70

Description:

The linked statement does not properly validate the returned `bool` of the **EIP-20** standard `transferFrom` function. As the **standard dictates**, callers **must not** assume that `false` is never returned.

Impact:

If the code mandates that the returned `bool` is `true`, this will cause incompatibility with tokens such as USDT / Tether as no such `bool` is returned to be evaluated causing the check to fail at all times. On the other hand, if the token utilized can return a `false` value under certain conditions but the code does not validate it, the contract itself can be compromised as having received / sent funds that it never did.

Example:

contracts/Migration.sol

SOL

```
70  IERC20(oldToken).transferFrom(msg.sender, address(this), balance);
```

Recommendation:

Since not all standardized tokens are **EIP-20** compliant (such as Tether / USDT), we advise a safe wrapper library to be utilized instead such as `SafeERC20` by OpenZeppelin to opportunistically validate the returned `bool` only if it exists.

Alleviation (bdef412a378a31cb3fb5fa519391b971ac7df48f):

The `SafeERC20` library from OpenZeppelin is imported and safely utilized in the referenced **EIP-20** transfer statement, addressing this exhibit in full.

PerpetualAirdropToken Static Analysis Findings

PAT-01S: Inexistent Sanitization of Input Address

Type	Severity	Location
Input Sanitization	Informational	PerpetualAirdropToken.sol:L29-L41

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/PerpetualAirdropToken.sol

SOL

```sol
29  constructor(
30      string memory tokenName,
31      string memory tokenSymbol,
32      uint8 _numLists,
33      address _migrationContract
34  ) ERC20(tokenName, tokenSymbol) ERC20Permit(tokenName) Ownable(msg.sender) {
35      numLists = _numLists;
36      for (uint8 i = 0; i < numLists; i++) {
37          eligibilityLists.push();
38          balanceThresholds.push(10 ** i * 1 ether);
```

Example (Cont.):

```sol
    }
    migrationContract = _migrationContract;
}
```

Recommendation:

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

Alleviation (bdef412a378a31cb3fb5fa519391b971ac7df48f):

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

TokenOne Static Analysis Findings

TOE-01S: Inexistent Sanitization of Input Address

Type	Severity	Location
Input Sanitization	Informational	TokenOne.sol:L86-L110

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/TokenOne.sol

SOL

```sol
constructor(
    string memory tokenName,
    string memory tokenSymbol,
    InitialAirdropConfig memory _initialAirdropConfig,
    RegularAirdropConfig memory _regularAirdropConfig,
    uint256 _governanceThreshold,
    address _migrationContract
) ERC20(tokenName, tokenSymbol) ERC20Permit(tokenName) Ownable(msg.sender) {
    require(_initialAirdropConfig.amount > 0, 'Invalid airdrop amount');
    require(_initialAirdropConfig.participants.length > 0, 'No participants');
```

Example (Cont.):

```sol
96      require(_regularAirdropConfig.amount > 0, 'Invalid regular airdrop amount');
97      require(_regularAirdropConfig.numWinners > 0, 'Invalid regular winners
count');
98      require(_regularAirdropConfig.airdropTimeInterval > 0, 'Invalid airdrop
interval');
99      require(
100         _regularAirdropConfig.windowDuration <=
_regularAirdropConfig.airdropTimeInterval,
101         'Invalid window duration'
102     );
103
104     initialAirdropConfig = _initialAirdropConfig;
105     regularAirdropConfig = _regularAirdropConfig;
106
107     governanceThreshold = _governanceThreshold;
108
109     migrationContract = _migrationContract;
110 }
```

Recommendation:

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

Alleviation (bdef412a378a31cb3fb5fa519391b971ac7df48f):

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

TokenOne Manual Review Findings

TOE-01M: Incorrect Voter Threshold Imposition

Type	Severity	Location
Logical Fault	Medium	TokenOne.sol:L467, L469

Description:

The `TokenOne::_updateTotalVotes` function will improperly impose the governance threshold as it will use a non-inclusive comparison for evaluating whether a user was previously a voter and an inclusive comparison for whether they are presently a voter.

This permits a user to stay at the same balance yet continuously increase the total vote checkpoint, permanently diluting the rewards distributed by the contract.

Impact:

It is possible to maliciously dilute all rewards of the `TokenOne` by staying at the `governanceThreshold` which the system would continuously treat as a movement from a non-voter to a voter state.

Example:

contracts/TokenOne.sol

SOL

```sol
465 function _updateTotalVotes(address account, uint256 currentBalance) internal {
466     uint256 previousBalance = voterBalances[account];
467     bool wasVoter = previousBalance > governanceThreshold;
468 
469     if (currentBalance >= governanceThreshold) {
470         if (!wasVoter) {
471             _totalVotesCheckpoints.push(clock(), SafeCast.toUint208(getTotalVotes()
+ currentBalance));
472         } else {
473             _totalVotesCheckpoints.push(clock(),
SafeCast.toUint208(getTotalVotes() + currentBalance - previousBalance));
474         }
```

Example (Cont.):

```sol
        voterBalances[account] = currentBalance;
    } else {
        if (wasVoter) {
            _totalVotesCheckpoints.push(clock(),
SafeCast.toUint208(getTotalVotes() - previousBalance));
        }
        voterBalances[account] = 0;
    }
}
```

Recommendation:

We advise the code to use an inclusive or non-inclusive comparison in both instances, alleviating this exhibit.

Alleviation (bdef412a378a31cb3fb5fa519391b971ac7df48f):

The conditional was made inclusive as advised, ensuring that the condition in which a voter is considered to be one matches the `wasVoter` variable declaration.

TOE-02M: Inexistent Update of Fulfilment Status

Type	Severity	Location
Logical Fault	Medium	TokenOne.sol:L310

Description:

The `TokenOne::fulfillRegularAirdrop` function will fail to set a particular airdrop as fulfilled if the last randomness fulfilment is executed with an empty eligibility list.

Impact:

In the unlikely event that the last randomness fulfilment of a regular airdrop is executed with an empty eligibility list, the airdrop's state will never move into the `Fulfilled` state preventing users from claiming their previously-distributed rewards.

Example:

contracts/TokenOne.sol

SOL

```sol
309 if (numParticipants == 0) {
310     state.numFulfilled += numRequested;
311     emit RegularAirdropFulfilled(requestId, index, numRequested);
312
313     return;
314 }
```

Recommendation:

We advise the code to properly validate whether the `numFulfilled` matches the `numWinners` of the airdrop configuration, and to set the `state.status` to `Fulfilled` in such a case.

Alleviation (bdef412a37):

The fulfilment status update has been incorrectly introduced as it does not ensure that the `numFulfilled` value matches the `numWinners` as advised, rendering the alleviation improper.

Alleviation (bef318aa99):

The code was updated per our original recommendation, ensuring that the state of the airdrop is set to a fulfilled one once the appropriate conditions are met.

PerpetualAirdropCoordinator Code Style Findings

PAC-01C: Inefficient Loop Limit Evaluation

Type	Severity	Location
Gas Optimization	Informational	PerpetualAirdropCoordinator.sol:L188

Description:

The linked `for` loop evaluates its limit inefficiently on each iteration.

Example:

contracts/PerpetualAirdropCoordinator.sol

SOL

```sol
188 for (uint256 i = 0; i < managedTokens.length(); i++) {
```

Recommendation:

We advise the statements within the `for` loop limit to be relocated outside to a local variable declaration that is consequently utilized for the evaluation to significantly reduce the codebase's gas cost. We should note the same optimization is applicable for storage reads present in such limits as they are newly read on each iteration (i.e. `length` members of arrays in storage).

Alleviation (bdef412a378a31cb3fb5fa519391b971ac7df48f):

The loop's length is evaluated outside its `for` loop body, optimizing its gas cost.

PAC-02C: Inefficient `mapping` Lookups

Type	Severity	Location
Gas Optimization	Informational	PerpetualAirdropCoordinator.sol:L495, L499, L503, L507

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

```sol
490 function distributeEarnings(
491     PerpetualAirdropToken token,
492     uint16 batchSize
493 ) public returns (uint16 numDistributed) {
494     if (batchSize == 0) {
495         batchSize = uint16(earnings[token].length());
496     }
497 
498     for (uint256 i = 0; i < batchSize; i++) {
499         if (earnings[token].length() == 0) {
```

Example (Cont.):

```sol
            break;
        }

        (address account, uint256 amount) = earnings[token].at(0);
        numDistributed++;

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

        // Emit event
        emit EarningDistributed(token, account, amount);

        // Mint tokens
        token.mint(account, amount);
    }
}
```

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 (bdef412a378a31cb3fb5fa519391b971ac7df48f):

All referenced inefficient `mapping` lookups have been optimized to the greatest extent possible, significantly reducing the gas cost of the functions the statements were located in.

PerpetualAirdropToken Code Style Findings

PAT-01C: Inefficient Iterator Type

Type	Severity	Location
Gas Optimization	Informational	PerpetualAirdropToken.sol:L36

Description:

The referenced iterator data type (`uint8`) is inefficient as it is less than 256 bits.

Example:

contracts/PerpetualAirdropToken.sol

SOL

```sol
36  for (uint8 i = 0; i < numLists; i++) {
```

Recommendation:

We advise a full 256 bit data type to be used (i.e. `uint256`) as the EVM is built to operate on 32-byte words and is thus more efficient when operating with them.

Alleviation (bdef412a378a31cb3fb5fa519391b971ac7df48f):

The referenced data type has been updated to `uint256`, optimizing its gas cost.

PAT-02C: Inefficient Loop Limit Evaluation

Type	Severity	Location
Gas Optimization	Informational	PerpetualAirdropToken.sol:L107

Description:

The linked `for` loop evaluates its limit inefficiently on each iteration.

Example:

contracts/PerpetualAirdropToken.sol

SOL

```sol
107 for (uint256 i = 0; i < transactionLoggers.length(); i++) {
```

Recommendation:

We advise the statements within the `for` loop limit to be relocated outside to a local variable declaration that is consequently utilized for the evaluation to significantly reduce the codebase's gas cost. We should note the same optimization is applicable for storage reads present in such limits as they are newly read on each iteration (i.e. `length` members of arrays in storage).

Alleviation (bdef412a378a31cb3fb5fa519391b971ac7df48f):

The loop's length is evaluated outside its `for` loop body, optimizing its gas cost.

TokenOne Code Style Findings

TOE-01C: Inefficient Calculations

Type	Severity	Location
Gas Optimization	Informational	TokenOne.sol:L396, L397, L399

Description:

The `remainder` calculation is inefficient as it will subtract the `topUp` from the total `amount` in an inefficient manner.

Example:

contracts/TokenOne.sol

SOL

```sol
396 uint256 remainder = amount - (state.computedAmount + topUp);
397 earnings.set(winner, currentAmount + topUp + remainder);
398
399 state.computedAmount += topUp + remainder;
```

Recommendation:

We advise the `topUp` to not be subtracted, permitting the resulting value to be utilized in place of the `topUp + remainder` calculations present in the code block.

Alleviation (bdef412a378a31cb3fb5fa519391b971ac7df48f):

The code was optimized as advised, reducing the gas cost involved in evaluating the last winner's earnings.

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.