OMNISCIA

August 4, 2023

SMART CONTRACT AUDIT REPORT

LightDoSo

Account Abstraction


omniscia.io


info@omniscia.io


Online report: lightdoso-account-abstraction

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

Account Abstraction Security Audit

Audit Report Revisions

Commit Hash	Date	Audit Report Hash
f76d46e5d8	July 26th 2023	539236c863
9574f5463f	August 4th 2023	474474430e

Audit Overview

We were tasked with performing an audit of the LightDotSo codebase and in particular their `LightWallet` implementation deriving from the Sequence Wallet and ETH Infinitism projects.

Over the course of the audit, we identified certain issues with the EIP-6492 signature validator that we advise the LightDotSo team to rectify as the contract presently fails to comply with the EIP standard.

As the `LightWallet` contract is meant to be deployed as a proxy that inherits from multiple external contracts, we investigated the codebases of the inherited Sequence Wallet and ETH Infinitism projects to ensure that no storage overlaps occur.

We confirmed that all data points in those external projects rely on `constant` and / or `immutable` variables, rendering them fully compatible between them as well as with the `LightWallet` implementation.

As a final evaluation, we assessed whether the `LightWallet` properly integrates the Sequence Wallet authorization mechanisms with the ETH Infinitism account abstraction system and validated that all necessary function signatures have been overridden properly.

We advise the LightDoSo 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 LightDoSo 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 LightDoSo and have identified that all exhibits have been adequately dealt with no outstanding issues remaining in the report.

Contracts Assessed

Files in Scope	Repository	Commit(s)
EntryPoint.sol (EPT)	LightDotSo	f76d46e5d8, 9574f5463f
ImmutableProxy.sol (IPY)	LightDotSo	f76d46e5d8, 9574f5463f
LightWallet.sol (LWT)	LightDotSo	f76d46e5d8, 9574f5463f
LightWalletUtils.sol (LWU)	LightDotSo	f76d46e5d8, 9574f5463f
LightWalletFactory.sol (LWF)	LightDotSo	f76d46e5d8, 9574f5463f
TokenPaymaster.sol (TPR)	LightDotSo	f76d46e5d8, 9574f5463f
UniversalSigValidator.sol (USV)	LightDotSo	f76d46e5d8, 9574f5463f

Audit Synopsis

Severity	Identified	Alleviated	Partially Alleviated	Acknowledged
Unknown	1	1	0	0
Informational	7	5	0	2
Minor	3	1	0	2
Medium	1	1	0	0
Major	0	0	0	0

During the audit, we filtered and validated a total of **2 findings utilizing static analysis** tools as well as identified a total of **10 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.

Compilation

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

To compile the project, the `build` command needs to be issued via the `forge` CLI tool:

```bash
forge build
```

The `forge` tool automatically selects Solidity version `0.8.18` based on the version specified within the `foundry.toml` file.

The project contains discrepancies with regards to the Solidity version used as the `pragma` statements of the contracts are open-ended (`^0.8.18`).

We advise them to be locked to `0.8.18` (`=0.8.18`), the same version utilized for our static analysis as well as optimizational review of the codebase.

During compilation with the `foundry` 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 **7 potential issues** within the codebase of which **4 were ruled out to be false positives** or negligible findings.

The remaining **3 issues** were validated and grouped and formalized into the **2 exhibits** that follow:

ID	Severity	Addressed	Title
LWF-01S	Minor	Yes	Inexistent Sanitization of Input Address
USV-01S	Informational	Yes	Multiple Top-Level Declarations

Manual Review

A **thorough line-by-line review** was conducted on the codebase to identify potential malfunctions and vulnerabilities in LightDoSo's account abstraction mechanism.

As the project at hand implements an account abstraction mechanism inheriting from multiple external projects, intricate care was put into ensuring that the **access control flows within the system conform to the specifications and restrictions** laid forth within each external protocol's specification and that **the storage spaces of all amalgamated contracts do not contain any harmful overlap**.

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 signature-related vulnerabilities** within the system's `UniversalSigValidator` contract which could have had **moderate ramifications** to its overall operation; we advise the LightDotSo team to prioritize the remediations of these vulnerabilities.

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 **10 findings** were identified over the course of the manual review of which **4 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
TPR-01M	Unknown	Nullified	Potentially Incorrect Usage of `TokenPaymaster`
USV-01M	Minor	Acknowledged	Inadequate Sanitization of Elliptic Curve Recovery Parameters
USV-02M	Minor	Acknowledged	Inexistent Validation of `ecrecover` Result
USV-03M	Medium	Yes	Deviation from EIP-6492 Standard

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
IPY-01C	Informational	Yes	Improper Compiler Warning Silencing
LWT-01C	Informational	Yes	Improper Compiler Warning Silencing
LWT-02C	Informational	Acknowledged	Loop Iterator Optimizations
LWF-01C	Informational	Yes	Generic Typographic Mistake
LWU-01C	Informational	Yes	Test Contract Implementation
USV-01C	Informational	Acknowledged	Redundant Self-Call

LightWalletFactory Static Analysis Findings

LWF-01S: Inexistent Sanitization of Input Address

Type	Severity	Location
Input Sanitization	Minor	LightWalletFactory.sol:L48-L50

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/src/LightWalletFactory.sol

SOL

```sol
48  constructor(IEntryPoint entryPoint) {
49      accountImplementation = new LightWallet(entryPoint);
50  }
```

Recommendation:

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

Alleviation:

The input `address` variable of the **`LightWalletFactory::constructor`** is now adequately sanitized as non-zero, preventing the contract from being misconfigured during its deployment and thus addressing this exhibit.

UniversalSigValidator Static Analysis Findings

USV-01S: Multiple Top-Level Declarations

Type	Severity	Location
Code Style	Informational	UniversalSigValidator.sol:L26, L33

Description:

The referenced file contains multiple top-level declarations that decrease the legibility of the codebase.

Example:

contracts/src/utils/UniversalSigValidator.sol

SOL

```
26  interface IERC1271Wallet {
27      function isValidSignature(bytes32 hash, bytes calldata signature) external
view returns (bytes4 magicValue);
28  }
29
30  error ERC1271Revert(bytes error);
31  error ERC6492DeployFailed(bytes error);
32
33  contract UniversalSigValidator {
```

Recommendation:

We advise all highlighted top-level declarations to be split into their respective code files, avoiding unnecessary imports as well as increasing the legibility of the codebase.

Alleviation:

The `IERC1271Wallet` interface (aptly renamed to `IERC1271`) has been relocated to its dedicated `IERC1271` file and is now imported to the `UniversalSigValidator` file properly, addressing this exhibit.

TokenPaymaster Manual Review Findings

TPR-01M: Potentially Incorrect Usage of `TokenPaymaster`

Type	Severity	Location
Standard Conformity	Unknown	TokenPaymaster.sol:L20

Description:

The `TokenPaymaster` of the LightDotSo repository simply imports the sample implementation of a `TokenPaymaster` by the `ETH` Infinitism project. This paymaster assumes a fixed `ETH` to token ratio that should be overridden by subclasses **per the official documentation of the contract**.

Impact:

The severity of this exhibit depends on whether the LightDotSo team intended to utilize the `ETH` Infinitism's sample fixed ratio of `100` between `ETH` and the token.

Example:

contracts/src/core/TokenPaymaster.sol

SOL

```sol
20  import "@eth-infinitism/account-abstraction/contracts/samples/TokenPaymaster.sol";
```

Recommendation:

We advise this trait to be evaluated and the `TokenPaymaster` to potentially expose a separate implementation that overrides the relevant function and utilizes either an owner-defined or oracle-assessed ratio between `ETH` and the token.

Alleviation:

The LightDotSo team re-evaluated the `TokenPaymaster` contract's purpose within their codebase and opted to eliminate it from the repository entirely as it is not presently used in their core contract suite. As such, we consider this exhibit nullified given that it is no longer relevant.

UniversalSigValidator Manual Review Findings

USV-01M: Inadequate Sanitization of Elliptic Curve Recovery Parameters

Type	Severity	Location
Language Specific	Minor	UniversalSigValidator.sol:L89

Description:

The `UniversalSigValidator::isValidSigImpl` will inadequately sanitize the `ecrecover` input arguments by ensuring that the provided `v` value is either `27` or `28` without validating the `s` value.

It is highly advisable to sanitize the `s` values due to the inherent trait of the elliptic curve in use by `ecrecover` to be symmetrical on the x-axis and thus permitting signatures to be replayed with the same `x` value (`r`) but a different `y` value (`s`).

Impact:

Most off-chain software will ensure that the generated `s` values lie within the lower-half of the ECDSA curve by themselves and this particular issue is rendered null by signature replay attacks that do not rely on the signature's hash, rendering the attack surface of this exhibit minimal and thus of "minor" severity.

Example:

contracts/src/utils/UniversalSigValidator.sol

SOL

```sol
85  require(_signature.length == 65, "SignatureValidator#recoverSigner: invalid signature length");
86  bytes32 r = bytes32(_signature[0:32]);
87  bytes32 s = bytes32(_signature[32:64]);
88  uint8 v = uint8(_signature[64]);
89  if (v != 27 && v != 28) {
90      revert("SignatureValidator: invalid signature v value");
91  }
92  return ecrecover(_hash, v, r, s) == _signer;
```

Recommendation:

We advise the `s` value to be sanitized, ensuring it is existent in the lower half order of the elliptic curve (`0 < s < secp256k1n ÷ 2 + 1`) by mandating it is less than `0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A1`. A reference implementation of those checks can be observed in the **ECDSA** library of OpenZeppelin and the rationale behind those restrictions exists within **Appendix F of the Yellow Paper**.

Alleviation:

The LightDotSo team stated that they have opted to acknowledge this exhibit to retain a one-to-one correlation of the `UniversalSigValidator` code and the example code of the official **EIP-6492**.

USV-02M: Inexistent Validation of `ecrecover` Result

Type	Severity	Location
Language Specific	Minor	UniversalSigValidator.sol:L92

Description:

A low-level `ecrecover` instruction will not yield an error for malformed signatures, instead returning an `address` of `0` as the result of the recovery operation.

Impact:

The susceptibility of a `UniversalSigValidator::isValidSigImpl` caller to this vulnerability depends on whether they in-turn validate that the intended `_signer` is not the zero-address. As such, we consider this exhibit to be of "minor" severity as most implementations guard against such entries and additionally prevent operations from them (such as OpenZeppelin's **EIP-20** implementation).

Example:

contracts/src/utils/UniversalSigValidator.sol

SOL

```
92  return ecrecover(_hash, v, r, s) == _signer;
```

Recommendation:

We advise the code to properly yield an error in case the `ecrecover` instruction result is the zero-address akin to all standard ECDSA signature validation implementations, such as **the Sequence** `SignatureValidator::recoverSigner` implementation, **the OpenZeppelin** `ECDSA::tryRecover` **implementation**, and more.

Alleviation:

The LightDotSo team stated that they have opted to acknowledge this exhibit to retain a one-to-one correlation of the `UniversalSigValidator` code and the example code of the official **EIP-6492**.

USV-03M: Deviation from EIP-6492 Standard

Type	Severity	Location
Standard Conformity	Medium	UniversalSigValidator.sol:L38, L56, L69, L80

Description:

The `UniversalSigValidator::isValidSigImpl` function is meant to support the **EIP-6492** standard, however, the code deviates from the standard in the way it validates counterfactual signatures with contract code that have failed.

In detail, the **EIP-6492** standard utilizes the `MUST` **RFC-2119** keyword to denote that signature verification should incorporate a "retry" with a forced call to the `create2Factory` with the `factoryCalldata` to prepare a potential proxy instance for **EIP-1271** signature validation.

Impact:

This exhibit describes a deviation of the `UniversalSigValidator` from the **EIP-6492** standard it is meant to implement under a special condition, warranting a severity of "medium".

Example:

contracts/src/utils/UniversalSigValidator.sol

SOL

```
38  function isValidSigImpl(address _signer, bytes32 _hash, bytes calldata _signature,
bool allowSideEffects)
39      public
40      returns (bool)
41  {
42      uint256 contractCodeLen = address(_signer).code.length;
43      bytes memory sigToValidate;
44      // The order here is striclty defined in https://eips.ethereum.org/EIPS/eip-
6492
45      // - ERC-6492 suffix check and verification first, while being permissive in
case the contract is already deployed; if the contract is deployed we will check the
sig against the deployed version, this allows 6492 signatures to still be validated
while taking into account potential key rotation
46      // - ERC-1271 verification if there's contract code
47      // - finally, ecrecover
```

Example (Cont.):

```sol
48      bool isCounterfactual =
49          bytes32(_signature[_signature.length - 32:_signature.length]) ==
ERC6492_DETECTION_SUFFIX;
50      if (isCounterfactual) {
51          address create2Factory;
52          bytes memory factoryCalldata;
53          (create2Factory, factoryCalldata, sigToValidate) =
54              abi.decode(_signature[0:_signature.length - 32], (address, bytes,
bytes));
55
56          if (contractCodeLen == 0) {
57              (bool success, bytes memory err) =
create2Factory.call(factoryCalldata);
58              if (!success) revert ERC6492DeployFailed(err);
59          }
60      } else {
61          sigToValidate = _signature;
62      }
63
64      // Try ERC-1271 verification
65      if (isCounterfactual || contractCodeLen > 0) {
66          try IERC1271Wallet(_signer).isValidSignature(_hash, sigToValidate)
returns (bytes4 magicValue) {
67              bool isValid = magicValue == ERC1271_SUCCESS;
68
69              if (contractCodeLen == 0 && isCounterfactual && !allowSideEffects) {
70                  // if the call had side effects we need to return the
71                  // result using a `revert` (to undo the state changes)
72                  assembly {
73                      mstore(0, isValid)
74                      revert(31, 1)
75                  }
```

Example (Cont.):

```sol
            }

            return isValid;
        } catch (bytes memory err) {
            revert ERC1271Revert(err);
        }
    }
```

Recommendation:

We advise the code to incorporate the retry-related code segments of the **EIP-6492** standard. Alternatively, we advise the code to clearly denote that it is incompatible with the **EIP-6492** standard due to the absence of a retry mechanism.

We consider either of those approaches as sufficient in rectifying this exhibit.

Alleviation:

The code of `UniversalSigValidator` was updated to properly conform to the **EIP-6492** standard and copy its logic, ensuring that forced "retry" calls are supported properly. As such, we consider this exhibit fully alleviated.

ImmutableProxy Code Style Findings

IPY-01C: Improper Compiler Warning Silencing

Type	Severity	Location
Gas Optimization	Informational	ImmutableProxy.sol:L28

Description:

The `ImmutableProxy::_authorizeUpgrade` function will contain an empty statement of the `newImplementation` argument to silence a compiler warning about unused function arguments, however, the syntax of Solidity permits function definitions without named arguments.

Example:

contracts/src/proxies/ImmutableProxy.sol

SOL

```sol
function _authorizeUpgrade(address newImplementation) internal pure override {
    (newImplementation);
    // Permanently disable upgrades by invalidating the authorization
    revert("Upgrades are disabled");
}
```

Recommendation:

We advise the `newImplementation` argument name to be omitted whilst retaining its type in the function signature of **`ImmutableProxy::_authorizeUpgrade`**, optimizing the code style of the code as well as the gas cost of the **`ImmutableProxy::_authorizeUpgrade`** function.

Alleviation:

The **`ImmutableProxy::_authorizeUpgrade`** function no longer contains an explicitly named argument, optimizing its code style as well as gas cost.

LightWallet Code Style Findings

LWT-01C: Improper Compiler Warning Silencing

Type	Severity	Location
Gas Optimization	Informational	LightWallet.sol:L180

Description:

The `LightWallet::_authorizeUpgrade` function will contain an empty statement of the `newImplementation` argument to silence a compiler warning about unused function arguments, however, the syntax of Solidity permits function definitions without named arguments.

Example:

contracts/src/LightWallet.sol

SOL

```sol
179 function _authorizeUpgrade(address newImplementation) internal view override
onlySelf {
180     (newImplementation);
181 }
```

Recommendation:

We advise the `newImplementation` argument name to be omitted whilst retaining its type in the function signature of **`LightWallet::_authorizeUpgrade`**, optimizing the code style of the code as well as the gas cost of the **`LightWallet::_authorizeUpgrade`** function.

Alleviation:

The **`LightWallet::_authorizeUpgrade`** function no longer contains an explicitly named argument, optimizing its code style as well as gas cost.

LWT-02C: Loop Iterator Optimizations

Type	Severity	Location
Gas Optimization	Informational	LightWallet.sol:L99, L103

Description:

The linked `for` loops increment / decrement their iterator "safely" due to Solidity's built - in safe arithmetics (post-`0.8.X`).

Example:

contracts/src/LightWallet.sol

SOL

```sol
99  for (uint256 i = 0; i < dest.length; i++) {
```

Recommendation:

We advise the increment / decrement operations to be performed in an `unchecked` code block as the last statement within each `for` loop to optimize their execution cost.

Alleviation:

The LightDotSo team evaluated this exhibit and opted to acknowledge it in the current iteration of the codebase.

LightWalletFactory Code Style Findings

LWF-01C: Generic Typographic Mistake

Type	Severity	Location
Code Style	Informational	LightWalletFactory.sol:L80

Description:

The referenced line contains a typographical mistake (i.e. `private` variable without an underscore prefix) or generic documentational error (i.e. copy-paste) that should be corrected.

Example:

contracts/src/LightWalletFactory.sol

SOL

```
80  // Computes the address with the given `salt`and the contract address
`addressImplementation`, and with `initialize` method w/ `owner`
```

Recommendation:

We advise this to be done so to enhance the legibility of the codebase.

Alleviation:

The typographic mistake referenced has been corrected, properly depicting the function and relevant argument of the initialization call.

LightWalletUtils Code Style Findings

LWU-01C: Test Contract Implementation

Type	Severity	Location
Code Style	Informational	LightWalletUtils.sol:L25

Description:

The `LightWalletUtils` contract represents a test contract meant to be utilized during the test suite of the LightDotSo project.

Example:

contracts/src/utils/LightWalletUtils.sol

SOL

```sol
20  import {Test} from "forge-std/Test.sol";
21
22  /// @title LightWalletUtils
23  /// @author shunkakinoki
24  /// @notice LightWalletUtils is a utility contract for the Safe
25  contract LightWalletUtils is Test {
```

Recommendation:

We advise it to be relocated to a different folder away from the `UniversalSigValidator` as the latter may be deployed in a production environment whereas the `LightWalletUtils` is expected to be utilized solely in a `forge` development environment.

Alleviation:

The `LightWalletUtils` contract has been relocated to a `test` sub-path of the project clearly indicating its intended usage and addressing this exhibit.

UniversalSigValidator Code Style Findings

USV-01C: Redundant Self-Call

Type	Severity	Location
Gas Optimization	Informational	UniversalSigValidator.sol:L99, L103

Description:

The referenced function invocation utilizes the `this` invocation syntax which will perform **an `external` call to the contract itself**.

Example:

contracts/src/utils/UniversalSigValidator.sol

SOL

```sol
95  function isValidSigWithSideEffects(address _signer, bytes32 _hash, bytes calldata
_signature)
96      external
97      returns (bool)
98  {
99      return this.isValidSigImpl(_signer, _hash, _signature, true);
100 }
```

Recommendation:

While no security implications arise from it, the `external` call performed is redundant and incurs an increased gas cost at no benefit.

We advise it to be replaced by a direct invocation of the `UniversalSigValidator::isValidSigImpl` function, omitting its `this.` prefix.

Alleviation:

The LightDotSo team stated that they have opted to acknowledge this exhibit to retain a one-to-one correlation of the `UniversalSigValidator` code and the example code of the official **EIP-6492**.

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.

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.