



Smart Contract Security Assessment

08-17-2022

Prepared for
Morpho

Online Report
[morpho-heap-ordering-structure](#)

Heap Ordering Structure Security Audit

Audit Overview

We were tasked with performing an audit of Morpho's data structures and in particular their `HeapOrdering` implementation.

The implementation is in fact a binary max-heap implementation with several optimizations to achieve the lowest operational cost when maintaining the structure's ordering integrity.

All indexes (contextually "ranks") are offset by `1` to ease bitwise operations and the structure implements the standard binary heap traversing operations by identifying parents and children using bitwise shift operations (i.e. a traversal upwards to the "parent" is performed using a right bitwise shift by `1` thus effectively achieving `[(i - 1) / 2]`).


We advise the Morpho 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 Morpho team addressed all exhibits identified in the report in the form of comprehensive PRs, dedicated issues opened in the main repository, and supplemental material provided in the PR and issue threads supporting their claims for nullification incurred by certain exhibits.

All exhibits have been adequately dealt with as evidenced in each exhibit's respective "Alleviation" chapter which provides a summary of the communications between as well as actions taken by both the Omniscia and Morpho team.

Contracts Assessed

Files in Scope	Repository	Commit(s)
HeapOrdering.sol (HOG)	data-structures 	6b3f07abe7, ba93e9d212, 0cd679077f

Audit Synopsis

Severity	Identified	Alleviated	Partially Alleviated	Acknowledged
 Unknown	1	1	0	0
 Informational	10	9	0	1
 Minor	1	0	0	1
 Medium	0	0	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 introduce potential misbehaviours of the system as well as exploits.

The list below covers each segment of the audit in depth and links to the respective chapter of the report:

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 `0.8.7` 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 (`^0.8.0`).

We advise them to be locked to `0.8.7` (`=0.8.7`), 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 **65 potential issues** within the codebase of which **61 were ruled out to be false positives** or negligible findings.

The remaining **4 issues** were validated and grouped and formalized into the **2 exhibits** that follow:

ID	Severity	Addressed	Title
HOG-01S	<div><div></div>Informational</div>	<div><div>✓</div>Yes</div>	Ineffectual Operations
HOG-02S	<div><div></div>Informational</div>	<div><div>✓</div>Yes</div>	Repetitive Value Literal

Manual Review

A **thorough line-by-line review** was conducted on the codebase to identify potential malfunctions and vulnerabilities in Morpho's binary max-heap implementation.

As the project at hand implements a specialized data structure, intricate care was put into ensuring that the **ordering integrity within the system is maintained across all operations** that are performed on it according to the specifications laid forth in the computer science academia.

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 potentially misusable functions** within the system which could have had **severe ramifications** to the structure's integrity if they were incorrectly utilized.

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 **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
HOG-01M	<div><div></div>Unknown</div>	<div><div>✓</div>Yes</div>	Potentially Ambiguous Functionality
HOG-02M	<div><div></div>Minor</div>	<div><div>!</div>Acknowledged</div>	Inexistent Association of Former Value

HeapOrdering Static Analysis Findings

HOG-01S: Ineffectual Operations

Type	Severity	Location
Gas Optimization	<div><div></div>Informational</div>	HeapOrdering.sol:L223-L224

Description:

The linked operations are ineffectual as they are performed on a `memory` reference that remains unutilized in the function body.

Example:

contracts/HeapOrdering.sol

SOL

```
210 /// @notice Increases the amount of an account in the `_heap`.
211 /// @dev Only call this function when `_id` is in the `_heap` with a smaller
212 /// @param _heap The heap to modify.
213 /// @param _id The address of the account to increase the amount.
214 /// @param _newValue The new value of the account.
215 /// @param _maxSortedUsers The maximum size of the heap.
216 function increase(
217     HeapArray storage _heap,
218     address _id,
219     uint256 _newValue,
220     uint256 _maxSortedUsers
221 ) private {
222     uint256 rank = _heap.ranks[_id];
223     Account memory account = getAccount(_heap, rank);
224     account.value = _newValue;
225     setAccountValue(_heap, rank, _newValue);
226     uint256 nextSize = _heap.size + 1;
227
228     if (rank < nextSize) shiftUp(_heap, rank);
229     else {
230         swap(_heap, nextSize, rank);
231         shiftUp(_heap, nextSize);
232         _heap.size = computeSize(nextSize, _maxSortedUsers);
233     }
```

```
233     }  
234 }
```

Recommendation:

We advise them to be safely omitted reducing the gas cost of the function.

Alleviation:

The redundant operations have been safely omitted from the codebase.

HOG-02S: Repetitive Value Literal

Type	Severity	Location
Code Style	Informational	HeapOrdering.sol:L128, L285, L304

Description:

The linked literal `1` is utilized in the linked portions to signify the head of the heap array, i.e. the root of the heap tree.

Example:

contracts/HeapOrdering.sol

SOL

```
297 /// @notice Returns the address coming before `_id` in accounts.
298 /// @dev The account associated to the returned address does not necessarily
299 /// @param _heap The heap to search in.
300 /// @param _id The address of the account.
301 /// @return The address of the previous account.
302 function getPrev(HeapArray storage _heap, address _id) internal view returns
303     uint256 rank = _heap.ranks[_id];
304     if (rank > 1) return getAccount(_heap, rank - 1).id;
305     else return address(0);
306 }
```

Recommendation:

We advise this to be better illustrated by setting a contract-level `constant` declaration (i.e. `HEAP_ROOT`) in which the value of `1` is stored greatly improving the legibility of the codebase. To note, in the first linked line only the first instance of `1` should be replaced as the latter instance is a binary offset.

Alleviation:

The issue has been completely alleviated by replacing all instances of the value literal with its `constant` declaration.

HeapOrdering Manual Review Findings

HOG-01M: Potentially Ambiguous Functionality

Type	Severity	Location
Standard Conformity	<div><div></div>Unknown</div>	HeapOrdering.sol:L289-L295

Description:

The `getTail` function yields the tail of the overall account array rather than the tail of the sorted heap array which can cause mis-use by the library's users.

Example:

contracts/HeapOrdering.sol

SOL

```
289 /// @notice Returns the address at the tail of the `_heap`.
290 /// @param _heap The heap to get the tail.
291 /// @return The address of the tail.
292 function getTail(HeapArray storage _heap) internal view returns (address) {
293     if (_heap.accounts.length > 0) return getAccount(_heap, _heap.accounts.
294     else return address(0);
295 }
```


Recommendation:

We advise a separate, dedicated `getHeapTail` function to be introduced that yields the ordered tree portion's tail member to its caller, increasing the utility of the library.

Alleviation:

The Morpho team opted to instead update the surrounding comments of the function better illustrating its purpose as the heap implementation in question is not meant for general use. As a result, we consider this exhibit adequately dealt with.

HOG-02M: Inexistent Association of Former Value

Type	Severity	Location
Input Sanitization	 Minor	HeapOrdering.sol:L30-L36

Description:

The `_formerValue` argument is meant to indicate the current value of the particular `_id` in the heap structure, however, this is not validated and is instead assumed to be trusted input.

Impact:

An improper `_formerValue` combined with an improper `_newValue` will cause an incorrect operation to be performed on the heap that does not match the actual data the `_id` entry previously held.

Example:

contracts/HeapOrdering.sol

SOL

```
30 function update(  
31     HeapArray storage _heap,  
32     address _id,  
33     uint256 _formerValue,  
34     uint256 _newValue,  
35     uint256 _maxSortedUsers  
36 ) internal {  
37     uint256 size = _heap.size;  
38     uint256 newSize = computeSize(size, _maxSortedUsers);  
39     if (size != newSize) _heap.size = newSize;  
40  
41     if (_formerValue != _newValue) {  
42         if (_newValue == 0) remove(_heap, _id, _formerValue);  
43         else if (_formerValue == 0) insert(_heap, _id, _newValue, _maxSortedUsers);  
44         else if (_formerValue < _newValue) increase(_heap, _id, _newValue, _maxSortedUsers);  
45         else decrease(_heap, _id, _newValue, _maxSortedUsers);  
46     }  
47 }
```

Recommendation:

While the contract is a `library` and thus input sanitization can be performed at the application level, from a security standpoint this reliance is ill-advised. Instead, we advise the `_formerValue` to be queried directly on the heap structure to guarantee all operations are performed safely regardless of end-user usage. If gas optimization is a concern here (as the data entry is already read by the caller), we advise a delta system to be used instead (i.e. specifying a decrease as `-5`) which would optimize the `if` conditionals and allow the entry to be read once at the heap level.

Alleviation:

The Morpho team evaluated this exhibit and deduced that the gas drawback of an additional `SLOAD` operation does not offset the sanitization benefit achieved as the library is meant for internal usage and Morpho expects its team to internally pass in only proper values. As a result, we consider this exhibit as acknowledged.

HeapOrdering Code Style Findings

HOG-01C: Downward Traversal Optimization

Type	Severity	Location
Gas Optimization	<div><div></div>Informational</div>	HeapOrdering.sol:L151, L154

Description:

The `shiftDown` downward traversal mechanism of the heap structure is sub-optimal as it computes the result of `getAccount` operations duplicate times in all cases.

Example:

contracts/HeapOrdering.sol

SOL

```
147 while (childRank <= size) {
148     // Compute the rank of the child with largest value.
149     if (
150         childRank < size &&
151         getAccount(_heap, childRank + 1).value > getAccount(_heap, childRank).value
152     ) childRank++;
153
154     childAccount = getAccount(_heap, childRank);
155
156     if (childAccount.value > initialValue) {
157         setAccount(_heap, _rank, childAccount);
158         _rank = childRank;
159         childRank <<= 1;
160     } else break;
161 }
```


Recommendation:

We advise the result of the first `getAccount` invocation within the `while` loop to be cached and overwritten in the `if` block by the next child's account lookup if the condition is met, thus ensuring `getAccount` is at most invoked a single time per child.

Alleviation:

The optimization has been applied as optimally as possible, enhancing the legibility of the codebase and optimizing its gas cost.

HOG-02C: Increase Swap Optimization

Type	Severity	Location
Gas Optimization	 Informational	HeapOrdering.sol:L230

Description:

The `swap` operation performed during an increase operation can be optimized by performing it conditionally only when `nextSize != rank` as otherwise the heap will replace an element with itself.

Example:

contracts/HeapOrdering.sol

SOL

```
210 /// @notice Increases the amount of an account in the `_heap`.
211 /// @dev Only call this function when `_id` is in the `_heap` with a smaller
212 /// @param _heap The heap to modify.
213 /// @param _id The address of the account to increase the amount.
214 /// @param _newValue The new value of the account.
215 /// @param _maxSortedUsers The maximum size of the heap.
216 function increase(
217     HeapArray storage _heap,
218     address _id,
219     uint256 _newValue,
220     uint256 _maxSortedUsers
221 ) private {
222     uint256 rank = _heap.ranks[_id];
223     Account memory account = getAccount(_heap, rank);
224     account.value = _newValue;
225     setAccountValue(_heap, rank, _newValue);
226     uint256 nextSize = _heap.size + 1;
227
228     if (rank < nextSize) shiftUp(_heap, rank);
229     else {
230         swap(_heap, nextSize, rank);
231         shiftUp(_heap, nextSize);
232         _heap.size = computeSize(nextSize, _maxSortedUsers);
233     }
234 }
```


Recommendation:

We advise this conditional to be introduced optimizing the best-case gas cost of the function with minimal impact on the worst-case scenario.

Alleviation:

The optimization has been properly applied at the `swap` function level instead of the calling code, ensuring that swaps are only performed as necessary.

HOG-03C: Insertion Swap Optimization

Type	Severity	Location
Gas Optimization	 Informational	HeapOrdering.sol:L188

Description:

The `swap` operation performed during an insertion operation can be optimized by performing it conditionally only when `newSize != accountsLength` as otherwise the heap will replace an element with itself.

Example:

contracts/HeapOrdering.sol

SOL

```
165 /// @notice Inserts an account in the `_heap`.
166 /// @dev Only call this function when `_id` is not in the `_heap`.
167 /// @dev Reverts with AddressIsZero if `_value` is 0.
168 /// @param _heap The heap to modify.
169 /// @param _id The address of the account to insert.
170 /// @param _value The value of the account to insert.
171 /// @param _maxSortedUsers The maximum size of the heap.
172 function insert(
173     HeapArray storage _heap,
174     address _id,
175     uint256 _value,
176     uint256 _maxSortedUsers
177 ) private {
178     // `_heap` cannot contain the 0 address.
179     if (_id == address(0)) revert AddressIsZero();
180
181     // Put the account at the end of accounts.
182     _heap.accounts.push(Account(_id, _value));
183     uint256 accountsLength = _heap.accounts.length;
184     _heap.ranks[_id] = accountsLength;
185
186     // Move the account at the end of the heap and restore the invariant.
187     uint256 newSize = _heap.size + 1;
188     swap(_heap, newSize, accountsLength);
189     shiftUp(_heap, newSize);
190     _heap.size = computeSize(newSize, _maxSortedUsers);
191 }
```

Recommendation:

We advise this conditional to be introduced optimizing the best-case gas cost of the function with minimal impact on the worst-case scenario.

Alleviation:

The optimization has been properly applied at the `swap` function level instead of the calling code, ensuring that swaps are only performed as necessary.

HOG-04C: Potential Significant Optimization

Type	Severity	Location
Gas Optimization	<div><div></div>Informational</div>	HeapOrdering.sol:L7

Description:

Depending on the needs of the application that will ultimately utilize the library, the `Account` data structure can be significantly optimized by reducing its `value` member's accuracy from 256-bits to 96-bits, thereby permitting the `address` variable to be tight-packed with the `uint96` into a single 32-byte slot. This will reduce all storage reads and writes for `Account` entries to one `SLOAD` / `SSTORE` operation regardless of whether both members are written to or read from in the same action.

Example:

contracts/HeapOrdering.sol

SOL

```
5  struct Account {
6      address id; // The address of the account.
7      uint256 value; // The value of the account.
8  }
```

Recommendation:

We advise this to be done so if the 96-bit accuracy is sufficient for the end-user application.

Alleviation:

The codebase has been updated to support the `uint96` data type significantly optimizing the gas cost of all operations due to the substantially less `SLOAD` operations required.

HOG-05C: Potential Significant Utility Optimization

Type	Severity	Location
Gas Optimization	<div><div></div>Informational</div>	HeapOrdering.sol:L4

Description:

The heap data structure implemented by the library is usually utilized to implement priority queues and other similar mechanisms where a common use case is to "pop" the upper-most parent of the "queue" and insert a new item that should be queued for the next iteration. If this is an envisioned use case for the heap, a significant utility and gas optimization can be achieved by implementing a dedicated "replace" function that replaces the root of the tree and then ascertains order by performing a single `shiftDown` operation.

Example:

contracts/HeapOrdering.sol

SOL

```
4 library HeapOrdering {
```

Recommendation:

We advise this use-case to be evaluated and the function described to be implemented in case it is envisioned as frequent given that it will lead to significant gas optimizations.

Alleviation:

The Morpho team evaluated this exhibit and concluded that the additional utility is not valuable to their use-case of the heap library and as such they will not proceed with implementing it. As a result, we consider this exhibit nullified given that it does not match Morpho's use case.

HOG-06C: Removal Swap Optimization

Type	Severity	Location
Gas Optimization	<div><div></div>Informational</div>	HeapOrdering.sol:L250

Description:

The `swap` operation performed during an removal operation can be optimized by performing it conditionally only when `rank != accountsLength` as otherwise the heap will replace an element with itself.

Example:

contracts/HeapOrdering.sol

SOL

```
236 /// @notice Removes an account in the `_heap`.
237 /// @dev Only call when this function `_id` is in the `_heap` with value `_removedValue`
238 /// @param _heap The heap to modify.
239 /// @param _id The address of the account to remove.
240 /// @param _removedValue The value of the account to remove.
241 function remove(
242     HeapArray storage _heap,
243     address _id,
244     uint256 _removedValue
245 ) private {
246     uint256 rank = _heap.ranks[_id];
247     uint256 accountsLength = _heap.accounts.length;
248
249     // Swap the last account and the account to remove, then pop it.
250     swap(_heap, rank, accountsLength);
251     if (_heap.size == accountsLength) _heap.size--;
252     _heap.accounts.pop();
253     delete _heap.ranks[_id];
254
255     // If the swapped account is in the heap, restore the invariant: its value must be greater than the value of the account it is swapped with.
256     if (rank <= _heap.size) {
257         if (_removedValue > getAccount(_heap, rank).value) shiftDown(_heap, rank);
258         else shiftUp(_heap, rank);
259     }
260 }
```


Recommendation:

We advise this conditional to be introduced optimizing the best-case gas cost of the function with minimal impact on the worst-case scenario.

Alleviation:

The optimization has been properly applied at the `swap` function level instead of the calling code, ensuring that swaps are only performed as necessary.

HOG-07C: Removal Upward Shift Optimization

Type	Severity	Location
Gas Optimization	 Informational	HeapOrdering.sol:L258

Description:

The `shiftUp` operation is performed unconditionally whilst a child and parent's value can indeed match, thus executing an upwards shift inefficiently if the element removed was next-to-last in the tree and its children had an equal value to it.

Example:

```
contracts/HeapOrdering.sol
SOL
255 // If the swapped account is in the heap, restore the invariant: its value
256 if (rank <= _heap.size) {
257     if (_removedValue > getAccount(_heap, rank).value) shiftDown(_heap, rank)
258     else shiftUp(_heap, rank);
259 }
```

Recommendation:

We advise the `else` statement to be adjusted to an `else if (_removedValue < getAccount(_heap, rank).value)` one, with the `getAccount(_heap, rank).value` value being cached to a local variable within the upper-most `if` clause to avoid the duplicate read gas cost between the `if` and `else if` conditional evaluations.

Alleviation:

The Morpho team stated that they do not believe an equality case to be frequent for their use-case of the heap structure and as such they will not proceed with applying the changes recommended by the exhibit instead acknowledging it.

HOG-08C: Upward Traversal Optimization

Type	Severity	Location
Gas Optimization	<div><div></div>Informational</div>	HeapOrdering.sol:L128, L129

Description:

The `shiftUp` upward traversal mechanism of the heap structure is sub-optimal as it computes the result of `getAccount(_heap, _rank >> 1)` twice redundantly.

Example:

contracts/HeapOrdering.sol

SOL

```
128 while (_rank > 1 && initialValue > getAccount(_heap, _rank >> 1).value) {
129     setAccount(_heap, _rank, getAccount(_heap, _rank >> 1));
130     _rank >>= 1;
131 }
```

Recommendation:

We advise the result to be cached to a local variable that is consequently utilized for the `while` loop condition and `setAccount` execution thus optimizing the gas cost of the function significantly.

Alleviation:

After extensive discussions surrounding the validity of the optimization, the Morpho team properly evaluated and assimilated the optimization described by this exhibit in the codebase ensuring that the upward traversal of the binary tree is performed as optimally as possible.

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.

External Call Validation

Many contracts that interact with DeFi contain a set of complex external call executions that need to happen in a particular sequence and whose execution is usually taken for granted whereby it is not always the case. External calls should always be validated, either in the form of `require` checks imposed at the contract-level or via more intricate mechanisms such as invoking an external getter-variable and ensuring that it has been properly updated.

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 Solidity language boasts that discerns it from other conventional programming languages. For example, the EVM is a 256-bit machine meaning that operations on less-than-256-bit types are more costly for the EVM in terms of gas costs, meaning that loops utilizing a `uint8` variable because their limit will never exceed the 8-bit range actually cost more than redundantly using a `uint256` variable.

Code Style

An official Solidity style guide exists that is constantly under development and is adjusted on each new Solidity release, designating how the overall look and feel of a codebase should be. 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 contract-level variable that is present in the inheritance chain of the local execution level's context.

Gas Optimization

Gas optimization findings relate to ways the codebase can be optimized to reduce the gas cost involved with interacting with it to various degrees. These types of findings are completely optional and are pointed out for the benefit of the project's developers.

Standard Conformity

These types of findings relate to incompatibility between a particular standard's implementation and the project's implementation, oftentimes causing significant issues in the usability of the contracts.

Mathematical Operations

In Solidity, math generally behaves differently than other programming languages due to the constraints of the EVM. A prime example of this difference is the truncation of values during a division which in turn leads to loss of precision and can cause systems to behave incorrectly when dealing with percentages and proportion calculations.

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.

Centralization Concern

This category covers all findings that relate to a significant degree of centralization present in the project and as such the potential of a Single-Point-of-Failure (SPoF) for the project that we urge them to re-consider and potentially omit.

Reentrant Call

This category relates to findings that arise from re-entrant external calls (such as EIP-721 minting operations) and revolve around the inapplicacy of the Checks-Effects-Interactions (CEI) pattern, a pattern that dictates checks (`require` statements etc.) should occur before effects (local storage updates) and interactions (external calls) should be performed last.

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.

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.