hashlock.

Security Audit

Vana (DeFi)

September 2024 info@hashlock.com.au Hashlock Pty Ltd

Table of Contents

CAUTION

THIS DOCUMENT IS A SECURITY AUDIT REPORT AND MAY CONTAIN CONFIDENTIAL INFORMATION. THIS INCLUDES IDENTIFIED VULNERABILITIES AND MALICIOUS CODE THAT COULD BE USED TO COMPROMISE THE PROJECT. THIS DOCUMENT SHOULD ONLY BE FOR INTERNAL USE UNTIL ISSUES ARE RESOLVED. ONCE VULNERABILITIES ARE REMEDIATED, THIS REPORT CAN BE MADE PUBLIC. THE CONTENT OF THIS REPORT IS OWNED BY HASHLOCK PTY LTD FOR THE USE OF THE CLIENT.

Executive Summary

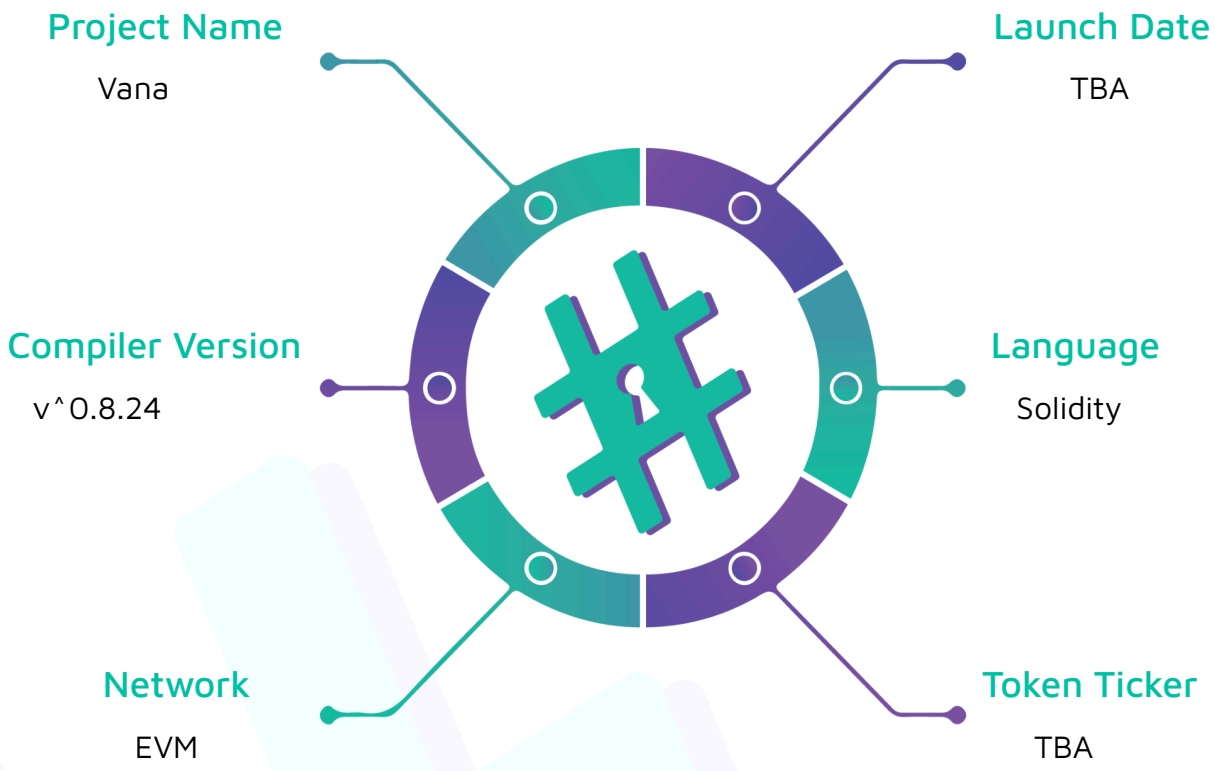
The Vana team partnered with Hashlock to conduct a security audit of their smart contracts. Hashlock manually and proactively reviewed the code to ensure the project's team and community that the deployed contracts were secure.

Project Context

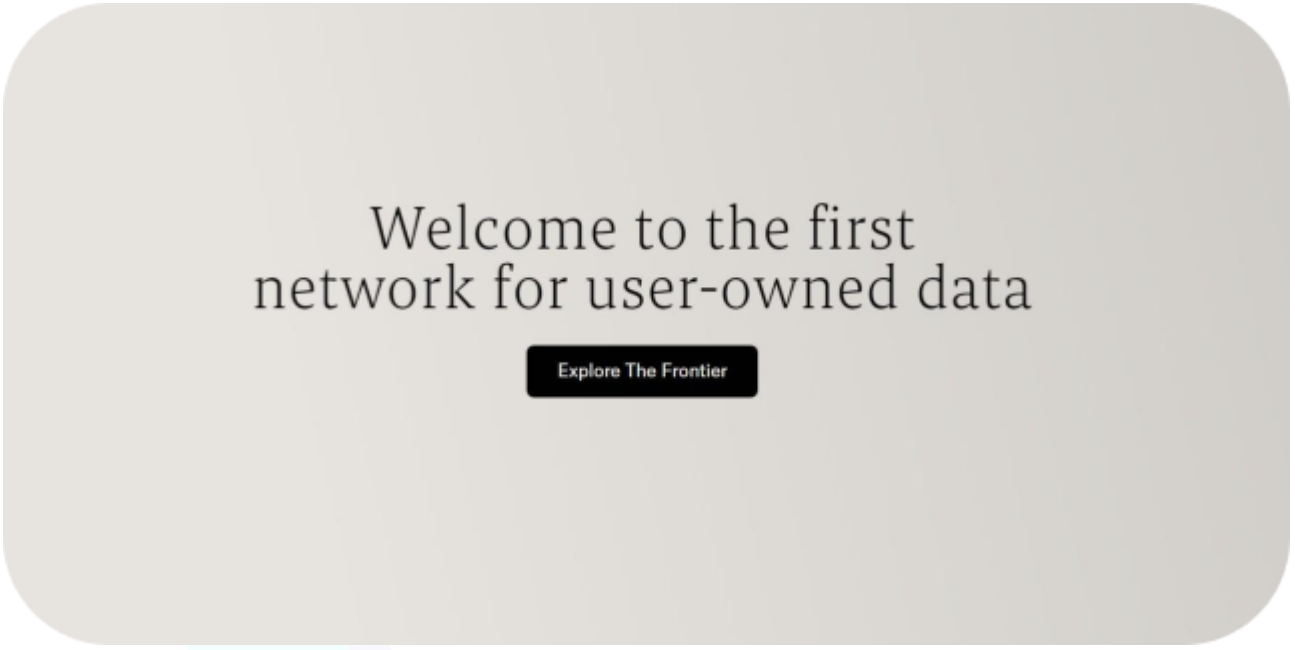
Vana is a distributed network for private, user-owned data, designed to enable user-owned AI. Users own, govern, and earn from the AI models they contribute to. Developers gain access to cross-platform data to power personalized applications and train frontier AI models.

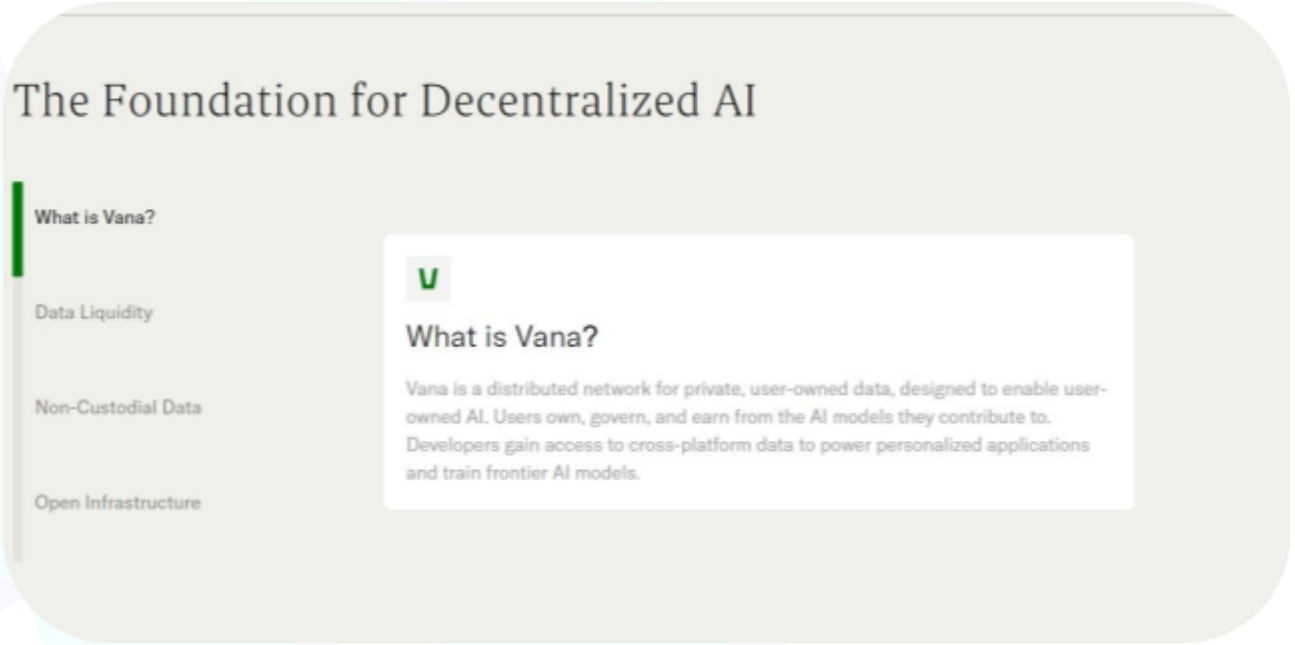
Project Name: Vana
Compiler Version: ^0.8.24
Website: www.vana.org
Logo:

Visualised Context:



Project Visuals:





Audit scope

We at Hashlock audited the solidity code within the Vana project, the scope of work included a comprehensive review of the smart contracts listed below. We tested the smart contracts to check for their security and efficiency. These tests were undertaken primarily through manual line-by-line analysis and were supported by software-assisted testing.

Note: Vana decided to clean up and copy the codebase into an entirely new repository for organization purposes. The content of the code of the deprecated repository and the new repository are identical after the fix review was conducted

Deprecated Repository Github Commit:
437d1506cd7b8af76cabc8b6a052ee919c737649
New Repository Github Commit:
5ffcfb40721b1ccc8e14bb843568f8711c3b779c

Description	Vana Smart Contracts
Platform	EVM / Solidity
Audit Date	September, 2024
Contract 1	DataRegistryImplementation.sol
Contract 1 MD5 Hash	48847d4c4daf1f84bf60a94ed2e637c7
Contract 2	DataLiquidityPoolImplementation.sol
Contract 2 MD5 Hash	b38b8ec8c72b0fadcdf801ffd80e1094
Contract 3	DataLiquidityPoolsRootImplementation.sol
Contract 3 MD5 Hash	68021757e827fe13b1fe6d927e3f1466
Contract 4	TeePoolImplementation.sol
Contract 4 MD5 Hash	d6a6d95790544fa1162973191fc3b20b
Contract 5	DAT.sol

Contract 5 MD5 Hash	c6cd96f9db6083919f31f6427a077dca
Contract 6	**TreasuryImplementation.sol**
Contract 6 MD5 Hash	2e148b341db534a108f60eae651356e2
Contract 7	**DepositImplementation.sol**
Contract 7 MD5 Hash	2e568a5599e984d38b9a5f219a867ecb
GitHub Commit Hash	437d1506cd7b8af76cabc8b6a052ee919c737649
GitHub Commit Hash v2	9d979d9df60b7aa9824209216aaf2c9faad7c0c4
GitHub Commit Hash v3	ec1f0f8fb069670db680bc08cbfebeb576c8480d
GitHub Commit Hash v4	5ffcfb40721b1ccc8e14bb843568f8711c3b779c

Security Rating

After Hashlock's Audit, we found the smart contracts to be **"Secure"**. The contracts all follow simple logic, with correct and detailed ordering.

The 'Hashlocked' rating is reserved for projects that ensure ongoing security via bug bounty programs or on-chain monitoring technology.

All issues uncovered during automated and manual analysis were meticulously reviewed and applicable vulnerabilities are presented in the Audit Findings section.

We initially identified some significant vulnerabilities that have since been addressed.

Hashlock found:

1 High-severity vulnerability

3 Medium-severity vulnerability

1 Low-severity vulnerabilities

3 QA

1 Gas Optimisation

Caution: *Hashlock's audits do not guarantee a project's success or ethics, and are not liable or responsible for security. Always conduct independent research about any project before interacting.*

Intended Smart Contract Behaviours

Claimed Behaviour	Actual Behaviour

DataRegistryImplementation.sol
- Allows users to:
  - Add a file to the registry
  - Adds a proof to the file
  - Add permissions for accounts to access the file
- Allows owner to:
  - Pause/unpause the contract
	Contract achieves this functionality.

DataLiquidityPoolImplementation.sol
- Allows users to:
  - Add a file
  - Add rewards for contributors
- Allows owner to:
  - Pause/unpause the contract
  - Update the file reward factor
  - Update the tee pool
  - Validate a file and send the contribution reward
  - Invalidate a file
	Contract achieves this functionality.

DataLiquidityPoolsRootImplementation.sol
- Allows users to:
  - Register/Deregister Dlps
  - Create epochs
  - Claim rewards
- Allows owner to:
  - Pause/unpause the contract
  - Update the maximum number of dlps
  - Update the epoch size and epoch reward amount
	Contract achieves this functionality.

- Update the minDlpStakeAmount and the performance percentages - Save the performances of top Dlps for a specific epoch	

TeePoolImplementation.sol
- Allows users to:
  - Request a contribution proof request
  - Submit/Cancel a job
  - Add a proof to the file
  - Claim rewards
- Allows owner to:
  - Pause/unpause the contract
  - Update the file registry
  - Update the tee fee
  - Update the cancel delay
  - Add/Remove a tee to the pool
	Contract achieves this functionality.

DAT.sol
- Allows users to:
  - Transfer tokens
- Allows owner to:
  - Mint tokens
  - Change admin address
  - Block mints
  - Block/Unblock addresses
	Contract achieves this functionality.

TreasuryImplementation.sol - Allows owner to: - Withdraw tokens	**Contract achieves this functionality.**
DepositImplementation.sol - Allows users to: - Deposit	**Contract achieves this functionality.**

- Allows owner to:
  - Update minDepositAmount/maxDepositAmount
  - Update the restricted
  - Add/Remove allowed validators	

Code Quality

This Audit scope involves the smart contracts of the Vana project, as outlined in the Audit Scope section. All contracts, libraries, and interfaces mostly follow standard best practices to help avoid unnecessary complexity that increases the likelihood of exploitation, however, some refactoring was required.

The code is very well commented on and closely follows best practice nat-spec styling. All comments are correctly aligned with code functionality.

Audit Resources

We were given the Vana projects smart contract code in the form of GitHub access.

As mentioned above, code parts are well-commented. The logic is straightforward, and therefore it is easy to quickly comprehend the programming flow as well as the complex code logic. The comments help understand the overall architecture of the protocol.

Dependencies

Per our observation, the libraries used in this smart contracts infrastructure are based on well-known industry-standard open-source projects.

Severity Definitions

Significance	Description
High	High-severity vulnerabilities can result in loss of funds, asset loss, access denial, and other critical issues that will result in the direct loss of funds and control by the owners and community.
Medium	Medium-level difficulties should be solved before deployment, but won't result in loss of funds.
Low	Low-level vulnerabilities are areas that lack best practices that may cause small complications in the future.
Gas	Gas Optimisations, issues, and inefficiencies

Audit Findings

High

[H-01] TeePoolImplementation#cancelJob - Lack of updating jobs bidAmount allows attackers to drain the contract

Description

The cancelJob function is missing from updating jobs[jobId].bidAmount and it allows anyone to call the function multiple times so they can drain the contract.

Vulnerability Details

The cancelJob function sends the native coins of jobs[jobId].bidAmount to the caller but jobs[jobId].bidAmount is not updated in the contract.

```
function cancelJob(uint256 jobId) external override {
    if (_jobs[jobId].ownerAddress != msg.sender) {
        revert NotJobOwner();
    }

    if (_jobs[jobId].addedTimestamp + cancelDelay > block.timestamp) {
        revert CancelDelayNotPassed();
    }

    payable(msg.sender).transfer(_jobs[jobId].bidAmount);

    _jobs[jobsCount].status = JobStatus.Canceled;

    emit JobCanceled(jobId);
}
```

Proof of Concept

An example of a test that simulates precision loss is shown below.

```
it("allow attackers to drain", async function () {

      const tx1 = await teePool

        .connect(user1)

        .requestContributionProof(1, { value: parseEther(0.01) });

      const tx2 = await teePool

        .connect(user2)

        .requestContributionProof(1, { value: parseEther(0.02) });

      let attackTx = await teePool

        .connect(attacker)

        .requestContributionProof(1, { value: parseEther(0.03) });



      const jobsCount = await teePool.jobsCount();

      await advanceNSeconds(cancelDelay);

      await advanceBlockNTimes(1);


      // attacker calls the function twice
      attackTx = await teePool.connect(attacker).cancelJob(jobsCount);

      attackTx = await teePool.connect(attacker).cancelJob(jobsCount);

      const contractBalance = await ethers.provider.getBalance(teePool.getAddress());
      (contractBalance).should.eq(0);
}
```

Impact

Attackers can transfer all the contract balance.

Recommendation

Update `jobs[jobId].bidAmount` to 0 before sending the bidAmount.

Status

Resolved

Medium

[M-01]

DataLiquidityPoolsRootImplementation#_createEpochsUntilBlockNumber - Unbounded loops when iterating over registered dlps in the topDlpIds function could cause out-of-gas error

Description

Because the _registerDlp function allows for an unlimited number of dlps to be registered, when the topDlpIds function iterates over all dlps registered, the contract may revert when a user tries to interact with the contract.

Vulnerability Details

The topDlpIds function loops over all dlps registered to get top dlps:

```
for (uint256 i = 0; i < registeredDlpsCount; i++) {
    ...
}
```

Depending on the number of dlps registered, calls to _createEpochsUntilBlockNumber may run out of gas, resulting in a denial of service of all external-facing functions.

Recommendation

It's recommended that the DataLiquidityPoolsRootImplementation sets a maximum number of dlps to be registered.

Status

Resolved

[M-02] DataRegistryImplementation, DataLiquidityPoolImplementation, DataLiquidityPoolsRootImplementation, TeePoolImplementation, TreasuryImplementation, DepositImplementation - Anyone can initialise the implementation contracts

Description

All contracts of the scope except for `DAT` are upgradeable smart contracts.

The implementation contracts allow anyone to call the `initialize()` function due to the absence of a `_disableInitializers()` call in the `constructor`.

Impact

Anyone can take control of the implementation contracts.

Recommendation

It's recommended to add `oz-upgrades-unsafe-allow constructor`.

```
/// @custom:oz-upgrades-unsafe-allow constructor

constructor() {

    _disableInitializers();

}
```

Status

Resolved

[M-03] DataLiquidityPoolsRootImplementation#estimatedDlpReward - Wrong assignment for the historyRewardEstimation return variable

Description

The historyRewardEstimation variable is intended to be based on the last epoch in which the dlp was part of.

But the current implementation assigns the historyRewardEstimation variable with the data based on the first epoch (which is the last iteration of the loop) in which the dlp was part of due to lack of break after the historyRewardEstimatio assignment.

Impact

Incorrect value is assigned to the historyRewardEstimatio return variable.

Recommendation

It's recommended to add a break after the historyRewardEstimatio assignment

```
for (uint256 i = epochsCount - 1; i >= lastEpochId; i--) {
      . . .
     historyRewardEstimation = (epochDlp.rewardAmount * epochDlp.stakersPercentage) /
epochDlp.stakeAmount;
     break;
}
```

Status

Resolved

Low

[L-01] DepositImplementation#updateMinDepositAmount, updateMaxDepositAmount, updateRestricted, addAllowedValidators, removeAllowedValidators - Lack of emitting events

Description

The updateMinDepositAmount, updateMaxDepositAmount, updateRestricted, addAllowedValidators, and removeAllowedValidators functions are missing events when key storages are updated.

Recommendation

Add relevant events based on the variables to be updated.

Status

Resolved

QA

[Q-01] DataLiquidityPoolsRootImplementation - Unused events and errors

Description

The `DataLiquidityPoolsRootImplementation` contract declared `DlpDeregisteredByOwner` and `ScoresUpdated` events but they are not used in the contract.

The same for `TooManyDlps` and `WithdrawNotAllowed` errors.

Recommendation

Remove unused events and errors.

Status

Resolved

[Q-02] TeePoolImplementation#canceljob - Incorrect function description

Description

The `cancelJob` function has the incorrect description in its comment.

Recommendation

Update the description with the correct one.

Status

Resolved

[Q-03] DataLiquidityPoolsRootImplementation, TeePoolImplementation - Deprecated Ether transfer function

Description

The `DataLiquidityPoolsRootImplementation` and `TeePoolImplementation` contracts use the `.transfer()` function to send the native coins.

However, the Istanbul update made some changes to the EVM, which made the `.transfer()` function deprecated for the ETH transfer.

Recommendation

Use `.call()` function to send ether instead of `.transfer()` function and check the return value.

Status

Resolved

Gas

[G-01] DataLiquidityPoolsRootImplementation#_registerDlp, DataRegistryImplementation#addProof, DataRegistryImplementation#_addFile - Use a memory instead of reading a storage multiple times

Description

The `_registerDlp` function reads `dlpCount` multiple times after it's updated.

The same for the `_files[fileId].proofsCount` in the `addProof` function and the `filesCount` in the `_addFile` function in the `DataRegistryImplementation` contract.

Recommendation

Add a local variable instead of reading from storage multiple times.

```
function _addFile(string memory url) internal {
    uint256 cachedFilesCount = filesCount++;
    _files[cachedFilesCount].ownerAddress = msg.sender;
    _files[cachedFilesCount].url = url;
    _files[cachedFilesCount].addedAtBlock = block.number;
    ...
}
```

Status

Resolved

Centralisation

The Vana project values security and utility over decentralisation.

The owner executable functions within the protocol increase security and functionality but depend highly on internal team responsibility.

Conclusion

After Hashlocks analysis, the Vana project seems to have a sound and well-tested code base now that our vulnerability findings have been resolved and acknowledged. Overall, most of the code is correctly ordered and follows industry best practices. The code is well commented on as well. To the best of our ability, Hashlock is not able to identify any further vulnerabilities.

Our Methodology

Hashlock strives to maintain a transparent working process and to make our audits a collaborative effort. The objective of our security audits is to improve the quality of systems and upcoming projects we review and to aim for sufficient remediation to help protect users and project leaders. Below is the methodology we use in our security audit process.

Manual Code Review:

In manually analysing all of the code, we seek to find any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our methodologies include manual code analysis, user interface interaction, and white box penetration testing. We consider the project's website, specifications, and whitepaper (if available) to attain a high-level understanding of what functionality the smart contract under review contains. We then communicate with the developers and founders to gain insight into their vision for the project. We install and deploy the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We undergo a robust, transparent process for analysing potential security vulnerabilities and seeing them through to successful remediation. When a potential issue is discovered, we immediately create an issue entry for it in this document, even though we still need to verify the feasibility and impact of the issue. This process is vast because we document our suspicions early even if they are later shown not to represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, and then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyse the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinised by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the contract details are made public.

Disclaimers

Hashlock's Disclaimer

Hashlock's team has analysed these smart contracts in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in the smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Hashlock is not responsible for the safety of any funds and is not in any way liable for the security of the project.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to attacks. Thus, the audit can't guarantee the explicit security of the audited smart contracts.

About Hashlock

Hashlock is an Australian-based company aiming to help facilitate the successful widespread adoption of distributed ledger technology. Our key services all have a focus on security, as well as projects that focus on streamlined adoption in the business sector.

Hashlock is excited to continue to grow its partnerships with developers and other web3-oriented companies to collaborate on secure innovation, helping businesses and decentralised entities alike.

Website: hashlock.com.au
Contact: info@hashlock.com.au


#Hashlock.