hashlock.

2024

Security Audit

Vana (DeFi)

November 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 WHICH 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 USE OF THE CLIENT.

Executive Summary

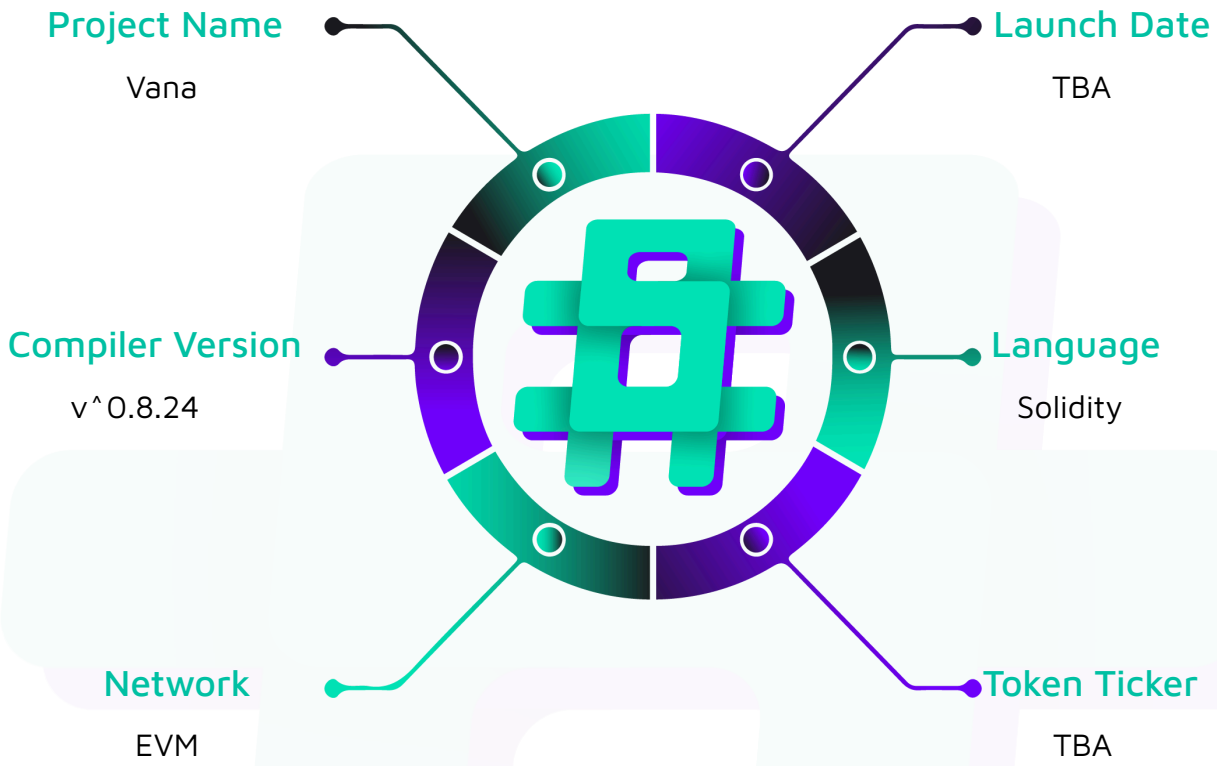
The Vana team partnered with Hashlock to conduct a security audit of their smart contracts. Hashlock manually and proactively reviewed the code to ensure the project's team and community that the deployed contracts were secure.

Project Context

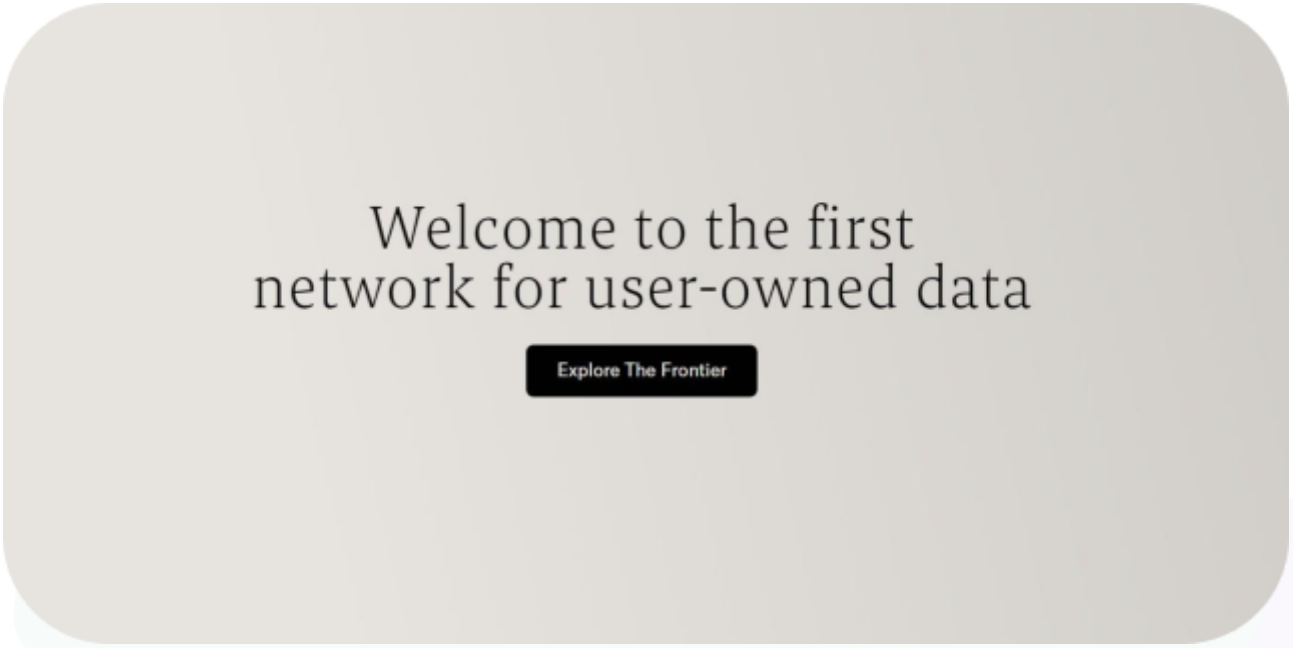
Vana is pioneering a transformative approach to data ownership and social impact by harnessing the power of Web3 and decentralized finance (DeFi). Through a groundbreaking ecosystem, individuals gain full sovereignty over their digital identities, turning data into an asset they truly own and control. By integrating DeFi principles, Vana enables users to monetize their data contributions via impact tokens, unlocking liquidity and new revenue streams within a decentralized marketplace. Unique NFT-backed data assets empower contributors with secure, tradable ownership, while DAO governance ensures the community collectively shapes the platform's evolution. Transparent smart contracts and a robust data commons framework create a circular economy where value flows seamlessly between personal gain and global impact. With Vana, your data becomes a catalyst for innovation, financial inclusion, and ethical transformation—redefining how individuals, communities, and economies thrive in the Web3 era.

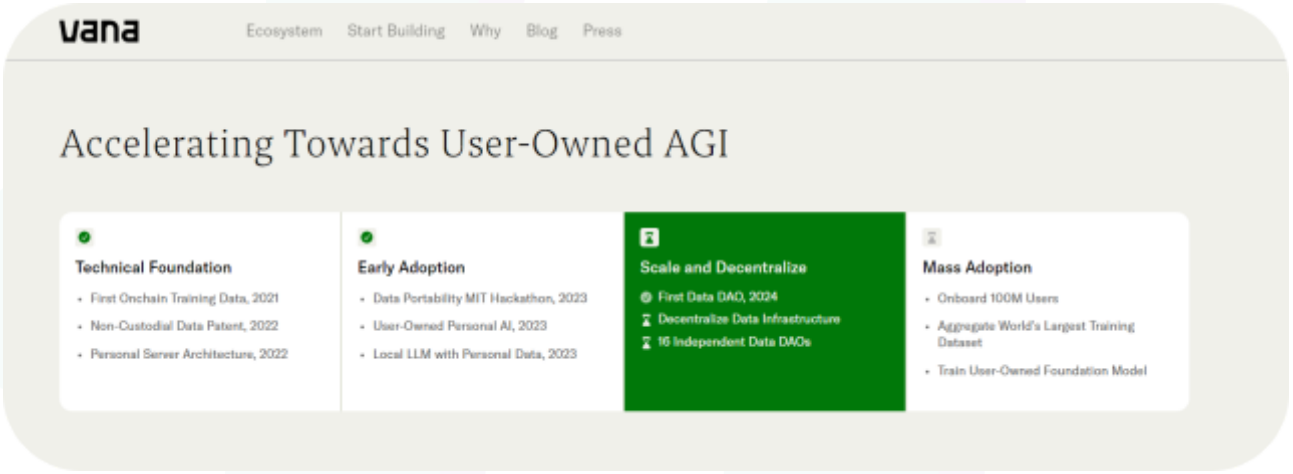
Project Name: Vana
Compiler Version: ^0.8.24
Website: www.vana.org
Logo:

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	**November, 2024**
Contract 1	**DataRegistryImplementation.sol**
Contract 1 MD5 Hash	c4e09439fdb53014605701e2844ee47d
Contract 2	**DataLiquidityPoolImplementation.sol**
Contract 2 MD5 Hash	d8acd418680bf8a86683e6f70e7b7a89
Contract 3	**DLPRootImplementation.sol**
Contract 3 MD5 Hash	e304d22e5ef1606e9ba7bb5ff697d384
Contract 4	**TeePoolImplementation.sol**
Contract 4 MD5 Hash	109b39f19bc368f94ea076e47bf5bfac
Contract 5	**MultisendImplementation.sol**
Contract 5 MD5 Hash	212a2fcc4177ce3c1e809f0ec2b5ecb1

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. They use a series of interfaces, and the protocol uses a list of Open Zeppelin contracts. We initially identified some significant vulnerabilities that have since been addressed.

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. The general security overview is presented in the Standardised Checks section and the project's contract functionality is presented in the Intended Smart Contract Functions section.

All vulnerabilities initially identified have now been resolved and acknowledged.

Hashlock found:

1 High severity vulnerability

1 Low severity vulnerability

1 Gas Optimisation

5 QAs

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 Functions

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 rewards for contributors
  - Request a reward for a file
- Allows owner to:
  - Pause/unpause the contract
  - Update the file reward factor
  - Update the tee pool
  - Update the proof instruction
  - Update the master key
	Contract achieves this functionality.

DLPRootImplementation.sol
- Allows users to:
  - Register/Deregister/Update Dlps
  - Create epochs
  - Claim rewards
  - Create/Close/Withdarw stakes
- Allows owner to:
  - Pause/unpause the contract
  - Update the eligible limit number of dlps
	Contract achieves this functionality.

- Update the min stake amount - Update the min dlp stakers percentage - Update the min dlp registration stake amount - Update eligibility threshold and adjust DLP statuses - Update sub-eligibility threshold and adjust DLP statuses - Update the epoch dlps limit - Update the stake withdrawal delay - Update the reward claim delay - Update the epoch size - Update the epoch reward amount - Update the stake scores for DLPs in past epochs	
TeePoolImplementation.sol - Allows users to: - Request a contribution proof request - Submit/Cancel a job - Add 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.**
MultisendImplementation.sol - Allows users to: - Send vanas to multiple recipients - Send tokens to multiple recipients	**Contract achieves this functionality.**

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 and 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 project 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 are helpful in providing an understanding of the protocol's overall architecture.

Dependencies

As 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
QA	Quality Assurance (QA) findings are purely informational and don't impact functionality. These notes help clients improve the clarity, maintainability, or overall structure of the code, ensuring a cleaner and more efficient project. They should be addressed for optimization but are not critical to the system's performance or security.

Audit Findings

High

[H-01] DataRegistryImplementation#addFileWithPermissions - Lack whenNotPaused modifier allows the function to be called in the paused status

Both adding files and permissions are not allowed in the paused status. However, users can bypass this restriction by using the `addFileWithPermissions` function.

Vulnerability Details

The `addFile` and `addFilePermission` functions don't allow users to add files and permissions in the paused status by having the `whenNotPaused` modifier.

```
function addFile(string memory url) external override whenNotPaused returns (uint256) {
    ...
}

function addFilePermission(uint256 fileId, address account, string memory key) external override whenNotPaused {
    ...
}
```

However, the `addFileWithPermissions` function doesn't have the `whenNotPaused` modifier and it allows users to add files and permissions in the paused status.

```
function addFileWithPermission(
    string memory url,
    address ownerAddress,
    Permission[] memory permissions
) external returns (uint256) {
    ...
}
```

Recommendation

Add the `whenNotPaused` modifier in the `addFileWithPermissions` function.

Status

Resolved

Low

[L-01] DataLiquidityPoolImplementation#updateTeePool - Lack of emitting an event

Description

The `updateTeePool` function is missing an event when the `teePool` variable is updated.

Recommendation

Add a relevant event in the function.

Status

Resolved

Gas

[G-01] DataRegistryImplementation#_addFile - Could return cachedFilesCount instead of filesCount

Description

In the `_addFile` function, the `filesCount` value is increased by 1 and then stored in the `cachedFilesCount` memory variable.

At the end of the function, it returns the `filesCount` value by reading the `filesCount` variable from storage even if the cached variable exists.

Recommendation

Return the `cachedFilesCount` instead of the `filesCount` value.

Status

Resolved

QA

[Q-01] DataLiquidityPoolImplementation#updateMasterKey - Incorrect parameter name and parameter description

Description

The `updateMaster` function is designed to update the `masterKey` variable but the function has an incorrect parameter name and parameter description.

Recommendation

Change the parameter name to `newMasterKey` and add a description for the parameter.

Status

Resolved

[Q-02] DLPRootImplementation - Unused event

Description

The `StakerDlpEpochRewardClaimed` event is not used in the contract.

Recommendation

Remove the unused event.

Status

Resolved

[Q-03] MultisendImplementation#multisendVana - Lack of an implementation to refund leftover could lead to the funds locked in the contract

Description

The `multisendVana` function is only passed when `msg.value` is equal to or greater than `amount * recipients.length`.

However, the function is missing an implementation to refund the leftover if `msg.value` is greater than `amount * recipients.length` and it could make the difference in the fund locked in the contract.

Recommendation

Calculate the difference between `msg.value` and `amount * recipients.length`, and if the difference is greater than 0, transfer the difference to the caller at the end of the function.

Status

Resolved

[Q-04] MultisendImplementation#multisendVana - Lack of return value check

Description

The `multisendVana` function transfers native coins with the `call` function.

```
function multisendVana(uint256 amount, address payable[] memory recipients) public payable nonReentrant {
    ...
    recipients[i].call{value: amount}(“”);
}
```

However, the lack of checking return values could make the function succeed even if the actual transfer failed.

Recommendation

Check if the return value is true, otherwise revert the function.

Status

Acknowledged

[Q-05] Contracts - Floating pragma

Description

The contracts use `pragma solidity ^0.8.24`.

This can allow the contracts to be deployed with the wrong pragma version which is different from the one the contracts were tested with.

Recommendation

Lock the pragma version.

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 behaviour 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 have not yet verified the feasibility and impact of the issue. This process is vast because we document our suspicions early even if they are later shown to not 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 as 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