hashlock.

Security Audit

World3.ai (DeFi)

July 2025 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

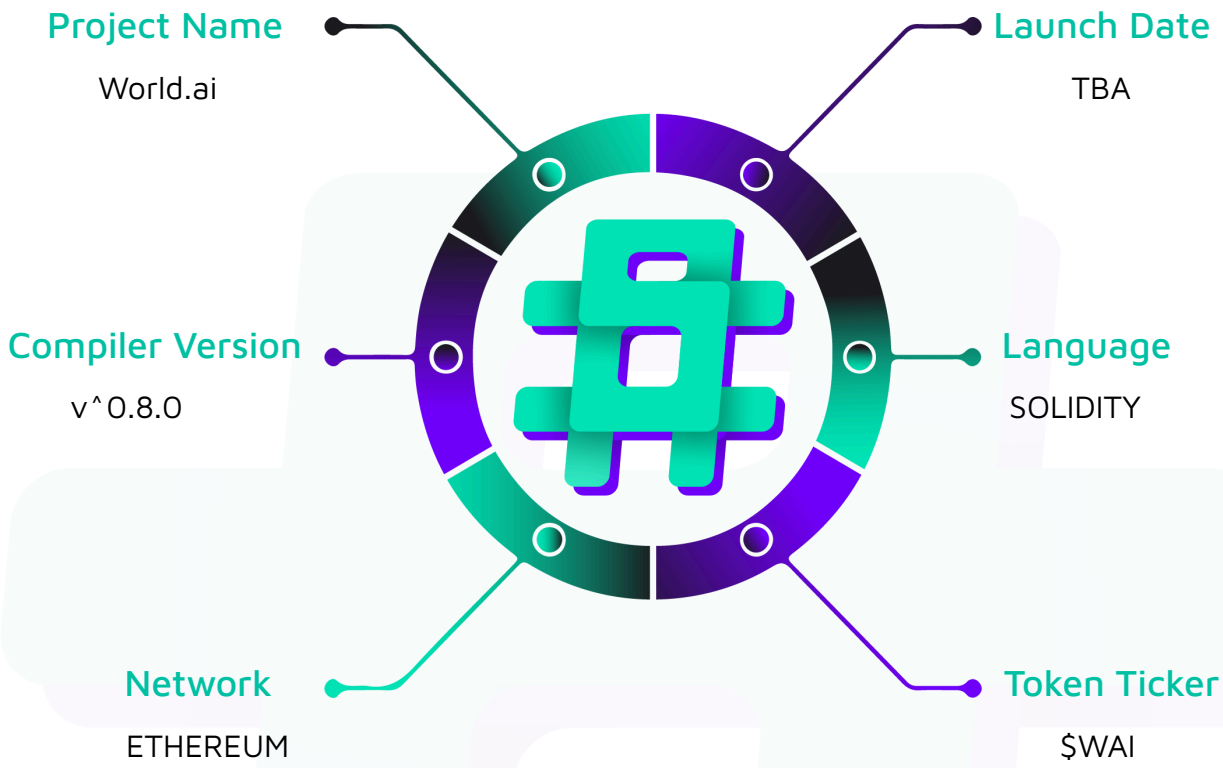
The World.ai team partnered with Hashlock to conduct a security audit of their smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

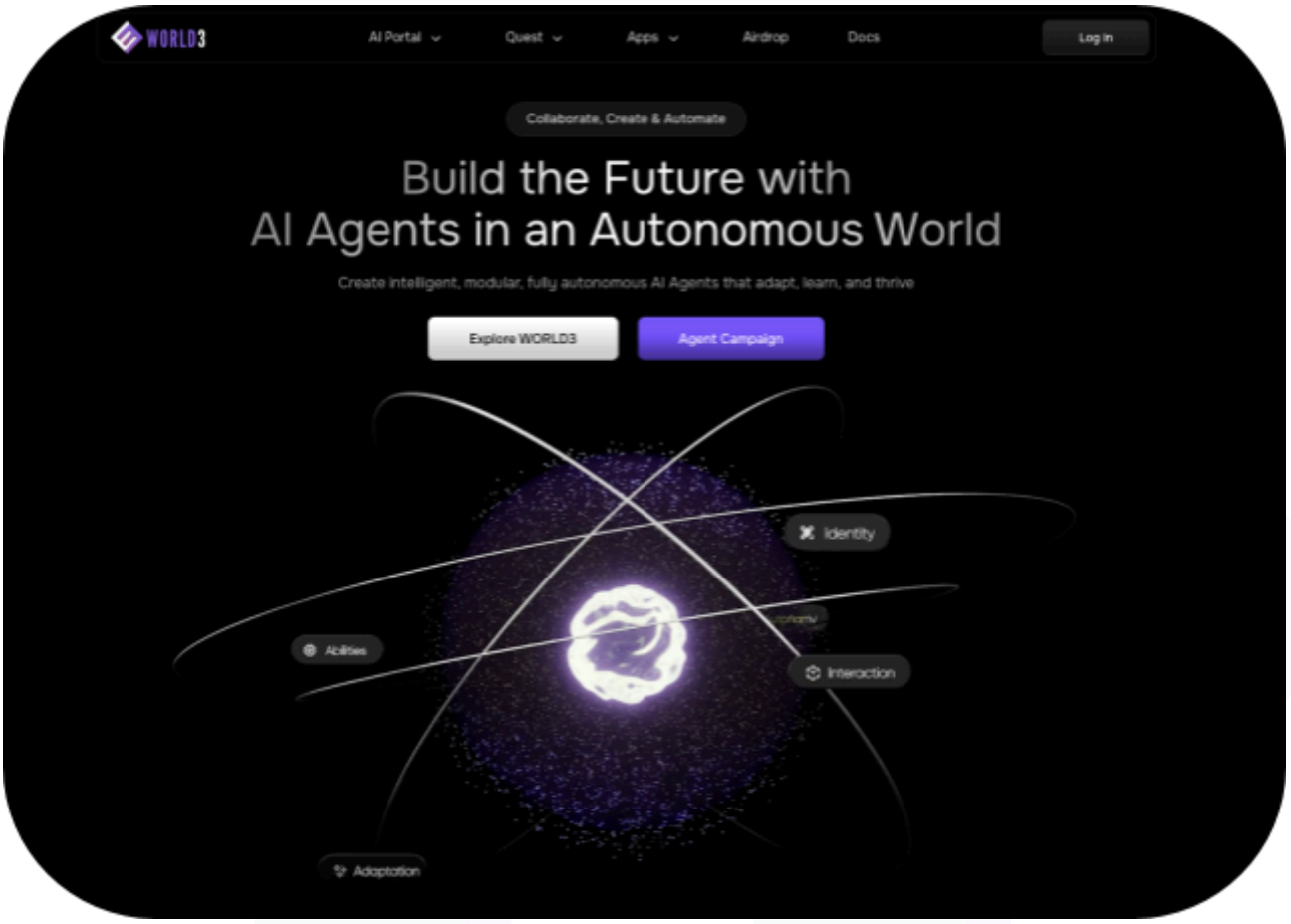
World.ai is an innovative Web3 platform that combines fully autonomous AI agents with blockchain technology to automate complex tasks 24/7. Based on the WORLD AI Protocol and supported by Azure (in collaboration with Microsoft for Startups), these agents can be created with no code using a drag-and-drop interface and configured through Skill Plugins and Knowledge Packs to handle tasks such as DeFi trading, DAO governance, social content creation, and interaction within blockchain games. The platform integrates decentralized identities, is multichain-compatible (including Bitcoin, SKALE, BNB, Aleo, and Flow), and utilizes its native token, $WAI, to pay for services and support an autonomous AI agent economy.

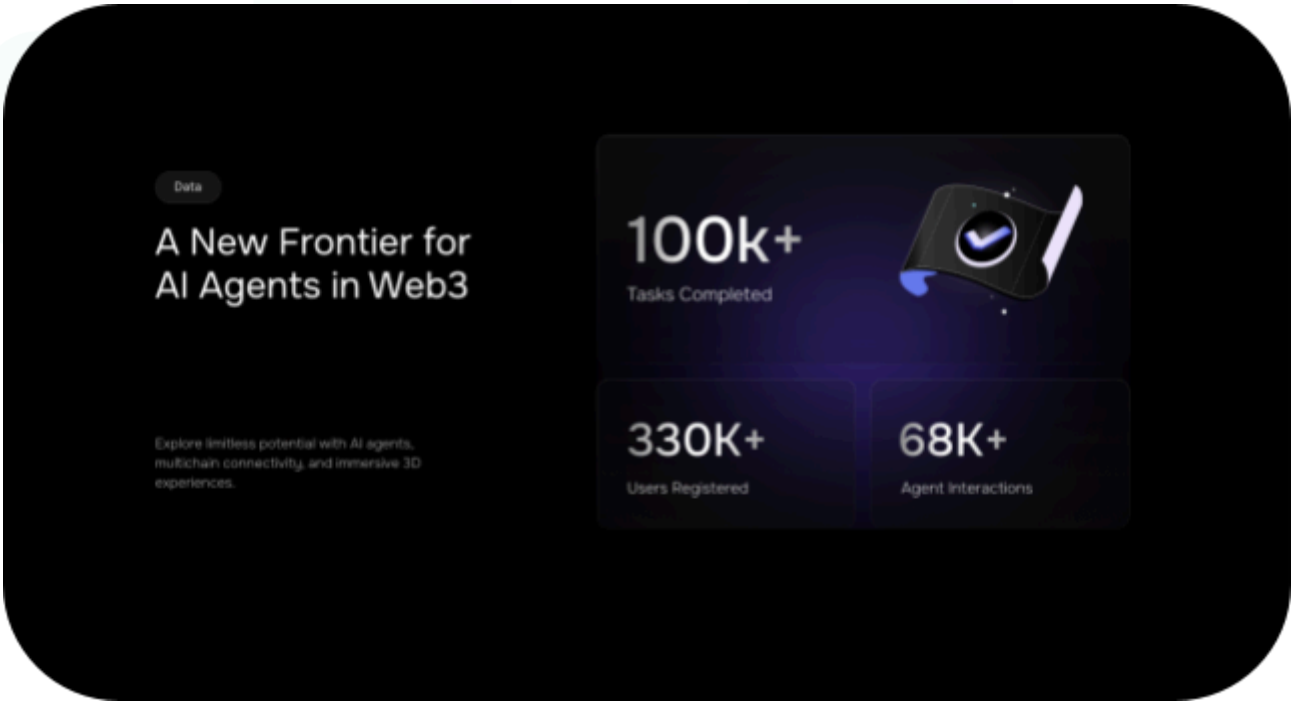
Project Name: World.ai
Project Type: DeFi
Compiler Version: ^0.8.0
Website: https://world3.ai/
Logo:

Visualised Context:



Project Visuals:





Audit Scope

We at Hashlock audited the solidity code within the World.ai 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.

Description	**World.ai Smart Contracts**
Platform	**Ethereum / Solidity**
Audit Date	**July, 2025**
Contract 1	WAIAirdrop.sol
Contract 2	wai.sol
Audited GitHub Commit Hash 1	50237018a390faae1de3e3f5a5c90dabafa27f85
Audited GitHub Commit Hash 2	cb48c12b4a6ef5aa1b85af8a8d84addc6d8969ed
Fix Review GitHub Commit Hash 1	54e111149e4e9f5180566b9ff19c5bd5bdfff08f
Fix Review GitHub Commit Hash 2	2e0610b1c52be44b5fa33e216b9ef293b46ad192

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.

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 list of audited assets is presented in the Audit Scope section and the project's contract functionality is presented in the Intended Smart Contract Functions section.

All vulnerabilities initially identified have now been resolved.

Hashlock found:

1 High severity vulnerability

1 Low severity vulnerability

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
WAIAirdrop.sol Allows users to: - Redeem their airdrop allocation using merkle proof validation - Claim all currently available vested tokens in a single transaction - Perform combined redeem and claim operations for better gas efficiency - Access all inherited airdrop and vesting functionality	**Contract achieves this functionality.**
wai.sol Allows owners to: - Pause/unpause token transfers while maintaining owner transfer privileges during pause - Recover accidentally sent ERC20 tokens and native tokens from the contract - Deploy a governance-enabled ERC20 token with voting power delegation capabilities	**Contract achieves this functionality.**

Code Quality

This audit scope involves the smart contracts of the World.ai 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 recommended to optimize security measures.

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 World.ai 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.
Apart from libraries, its functions are used in external smart contract calls.

Severity Definitions

The severity levels assigned to findings represent a comprehensive evaluation of both their potential impact and the likelihood of occurrence within the system. These categorizations are established based on Hashlock's professional standards and expertise, incorporating both industry best practices and our discretion as security auditors. This ensures a tailored assessment that reflects the specific context and risk profile of each finding.

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 informational and don't impact functionality. Supports clients improve the clarity, maintainability, or overall structure of the code.

Status Definitions

Each identified security finding is assigned a status that reflects its current stage of remediation or acknowledgment. The status provides clarity on the handling of the issue and ensures transparency in the auditing process. The statuses are as follows:

Significance	Description
Resolved	The identified vulnerability has been fully mitigated either through the implementation of the recommended solution proposed by Hashlock or through an alternative client-provided solution that demonstrably addresses the issue.
Acknowledged	The client has formally recognized the vulnerability but has chosen not to address it due to the high cost or complexity of remediation. This status is acceptable for medium and low-severity findings after internal review and agreement. However, all high-severity findings must be resolved without exception.
Unresolved	The finding remains neither remediated nor formally acknowledged by the client, leaving the vulnerability unaddressed.

Audit Findings

High

[H-01] wai#_update - Double-Accounting Bug in _update Function When Paused

Description

When the contract is paused, any transfer initiated by the contract owner triggers a double-spending bug. This results in the sender's account being debited for twice the intended transfer amount, and the recipient's account being credited for twice the amount.

Vulnerability Details

The vulnerability exists in the `_update` function's logic for owner transfers during a paused state. The function incorrectly calls both `ERC20._update` and `ERC20Votes._update` directly. The `ERC20Votes._update` function itself contains a `super._update()` call, which resolves back to `ERC20._update`. This sequence causes the core balance-update logic in `ERC20` to be executed twice for a single transaction.

```
function _update(
        address from,
        address to,
        uint256 value
    ) internal override(ERC20, ERC20Votes, ERC20Pausable) {
        // Override the pausable check to allow owner transfers when paused
        if (paused()) {
            // Only check pause for non-owner
            if (msg.sender != owner()) {
                revert EnforcedPause();
```

```
            }

            // Owner can transfer even when paused, so skip the normal pause check

            // by calling the parent implementations directly

            ERC20._update(from, to, value);

            ERC20Votes._update(from, to, value);

        } else {

            // When not paused, use normal flow

            super._update(from, to, value);

        }

    }
```

Proof of Concept

An example of a test that simulates `DoubleAccounting` is shown below.

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.22;



import {Test} from "forge-std/Test.sol";

import {WORLD3AICoin} from "../contracts/wai.sol";



contract DoubleAccountingPOCTest is Test {

    WORLD3AICoin public wai;

    address public owner;

    address public recipient;



    uint256 public constant INITIAL_SUPPLY = 1_000_000_000 * 10**18;

    uint256 public constant TRANSFER_AMOUNT = 50_000_000 * 10**18;
```

```
    function setUp() public {
        // Create an owner and a recipient for the test
        owner = makeAddr("owner");
        recipient = makeAddr("recipient");

        // Deploy the contract, making the 'owner' address the multisig wallet
        vm.prank(owner);
        wai = new WORLD3AICoin(owner);
    }

    /**
     * @notice This test demonstrates the double-accounting bug.
     *
     * 1. The contract is paused by the owner.
     * 2. The owner (acting as multisig) transfers 50 million tokens.
     * 3. Because of the bug in _update(), the balance is updated twice.
     * 4. The owner's balance decreases by 100 million instead of 50 million.
     * 5. The recipient's balance increases by 100 million instead of 50 million.
     */
    function test_poc_double_accounting_bug() public {
        // --- Get Initial Balances ---
        uint256 ownerInitialBalance = wai.balanceOf(owner);
        uint256 recipientInitialBalance = wai.balanceOf(recipient);

        // --- Verify Initial State ---
        // Owner should have the full initial supply
```

```
        assertEq(ownerInitialBalance, INITIAL_SUPPLY, "Owner initial balance should be the total supply");
        // Recipient should have no tokens
        assertEq(recipientInitialBalance, 0, "Recipient initial balance should be zero");

        // --- Trigger the Bug ---
        // 1. Owner pauses the contract
        vm.prank(owner);
        wai.pause();
        assertTrue(wai.paused(), "Contract should be paused");

        // 2. Owner transfers tokens while the contract is paused
        vm.prank(owner);
        wai.transfer(recipient, TRANSFER_AMOUNT);

        // --- Verify Final Balances ---
        uint256 ownerFinalBalance = wai.balanceOf(owner);
        uint256 recipientFinalBalance = wai.balanceOf(recipient);

        // --- Assert the Bug's Effect ---
        // Owner's balance should have decreased by TWICE the transfer amount
        assertEq(
            ownerFinalBalance,
            ownerInitialBalance - (TRANSFER_AMOUNT * 2),
            "BUG: Owner's balance was debited twice"
        );
```

```
        // Recipient's balance should have increased by TWICE the transfer amount
        assertEq(
            recipientFinalBalance,
            recipientInitialBalance + (TRANSFER_AMOUNT * 2),
            "BUG: Recipient's balance was credited twice"
        );

        // --- Show the correct expected outcome for comparison ---
        // This assertion will fail, but it shows what the balance *should* have been.
        assertEq(
            ownerFinalBalance,
            ownerInitialBalance - TRANSFER_AMOUNT,
            "CORRECT (will fail): Owner balance should decrease by transfer amount"
        );
    }
}
```

Impact

When the contract is paused, any transfer by the owner will cause the sender to lose double the intended amount and the recipient to receive double. This flaw breaks the token's fundamental accounting, leading to direct financial loss.

Recommendation

The `_update` function should be modified to ensure the balance-update logic is only ever called once.

Status

Resolved

Low

`[L-01] Contracts - Use of floating pragma

Description

The contracts WAIAirdrop.sol and Wai.sol (uses ^0.8.22) use a floating pragma version (`>=0.8.0 <0.9.0`). It's crucial to compile and deploy contracts with the same compiler version and settings used during development and testing. Fixing the pragma version prevents compilation with different versions. Using an outdated pragma version could introduce bugs that negatively affect the contract system, while newly released versions may have undiscovered security vulnerabilities.

Recommendation

Change the pragma statement in all contracts to a fixed version, such as `pragma solidity 0.8.0`. This locks the contract to a specific compiler version.

Status

Resolved

Centralisation

The World.ai 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 Hashlock's analysis, the World.ai project seems to have a sound and well-tested code base, now that our vulnerability findings have been resolved. 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