hashlock.

Security Audit

OpenZK

February 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

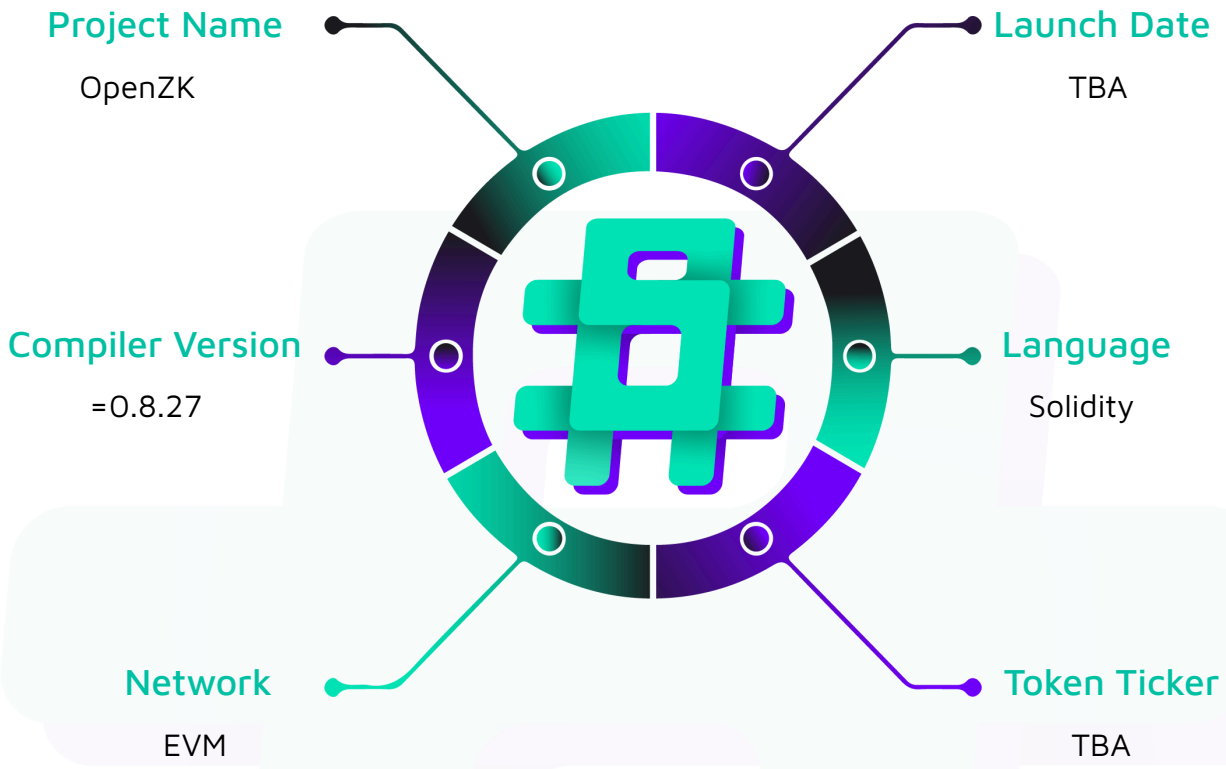
The OpenZK team partnered with Hashlock to conduct a security audit of their SkyMoneyVault.sol smart contract. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

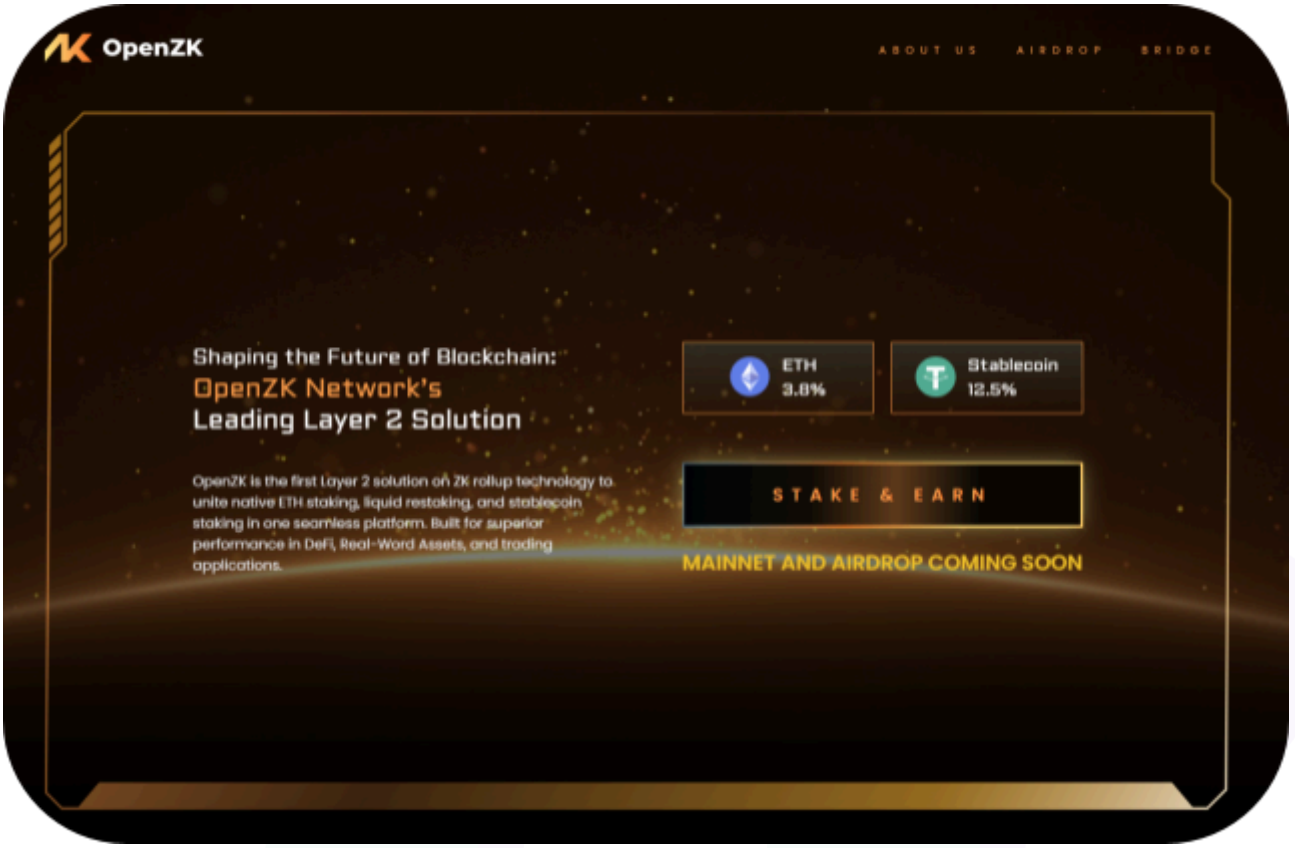
OpenZK is an innovative Layer 2 (L2) network project that leverages Zero-Knowledge (ZK) Rollup technology to enhance Ethereum's scalability and transaction efficiency while maintaining high security. By integrating native ETH staking and restaking capabilities, OpenZK not only bolsters scalability but also creates new revenue streams for users. The platform's leadership includes co-founder Dave Sandor, formerly an Executive Director at Goldman Sachs Asia-Pacific, co-founder and CTO Lucas Cullen, an early contributor to Ethereum's development, and also co-founder Jenna Wayne. OpenZK distinguishes itself with a Dual Gas Fee Mechanism, allowing users to pay gas fees with its native and protocol tokens, enhancing network flexibility and creating sustained demand for its protocol token.

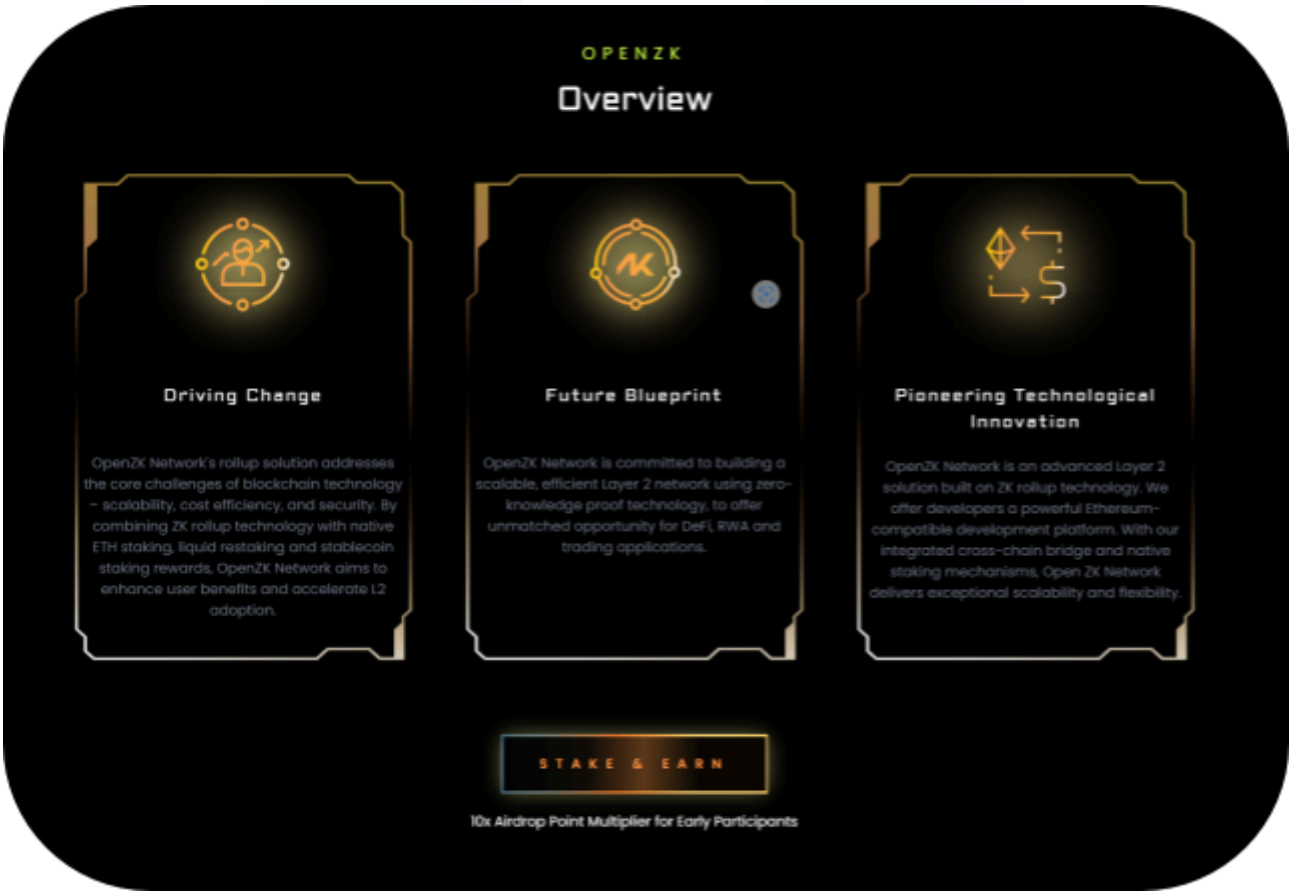
Project Name: OpenZK
Project Type: DeFi
Compiler Version: =0.8.27
Website: www.openzk.net
Logo:

Visualised Context:



Project Visuals:





Audit scope

We at Hashlock audited the solidity code within the OpenZK 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	OpenZK Smart Contract
Platform	Ethereum / Solidity
Audit Date	February, 2025
Contract 1	SkyMoneyVault.sol
Contract 1 MD5 Hash	3f8901445052672b19231d6a5f61d48c
Github Commit Hash	7fddc2fa7237f013d77a2fdaa9c1af46257733ce

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:

2 High severity vulnerabilities

4 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

SkyMoneyVault.sol
- Allows users to:
  - Deposit any tokens and get shares
  - Withdraw shares
- Allows owner to:
  - Set a new cliff
	Contract achieves this functionality.

Code Quality

This audit scope involves the smart contracts of the OpenZK 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 OpenZK 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

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] SkyMoneyVault#queueWithdrawToken - Incorrect token conversion

Description

The contract allows users to withdraw their shares into any tokens and users can set the token address that they want to receive in the `queueWithdrawToken` function.
The `queueWithdrawToken` function internally withdraws the shares to the underlying tokens first and then swaps the underlying tokens withdrawn to the tokens which users want to receive.

But the current implementation is swapping the tokens in the opposite way by passing parameters with incorrect order to the `swapWithMaximumSlippage` function.

Vulnerability Details

In the `swapWithMaximumSlippage` function, the 3rd parameter is supposed to be the input token and 4th parameter is supposed to be the output token.
But even though the contract is supposed to receive tokens of `token` address, the current `queueWithdrawToken` function is swapping the tokens in the opposite way by passing the `token` address as the input token parameter and `_underlying` address as the output token parameter.

```
function queueWithdrawToken(
    uint256 shares,
    address token
) external returns (uint256 index, uint256 cliff) {
    ...
    uint256 amountSwapped = Swap.swapWithMaximumSlippage(
        _univ3router,
        _fee,
        token,
        _underlying,
```

```
        _self,
        dai
    );
    ...
}
```

Impact

The transaction always fails since the contract doesn't hold tokens of the `token` address.

Recommendation

Pass the token addresses with the correct order to the `swapWithMaximumSlippage` and add the underlying tokens approval to the router contract before `swapWithMaximumSlippage` call.

Status

Resolved

[H-02] Swap#swapWithMaximumSlippage - Incorrect slippage calculation

Description

The contract allows users to withdraw their shares into any tokens and users can set the token address that they want to receive in the `queueWithdrawToken` function.
The `queueWithdrawToken` function internally calls `Swap.swapWithMaximumSlippage` function and the minimum amount of output tokens is set to 97% of the input token amount in `swapWithMaximumSlippage` function.

However, the minimum amount is calculated assuming that both input and output tokens prices are the same and also it doesn't consider the decimals of the input token and output token.

Vulnerability Details

In the `swapWithMaximumSlippage` function, `amountOutMinimum` variable is updated to 97% of `amountIn` value.

However, `amountOutMinimum` value is the amount of output token and `amountIn` value is the amount of input token.

The current `amountOutMinimum` calculation is incorrect since it's calculated regardless of the token prices.

Even though both tokens have the same prices, if the decimals of both tokens are different, `amountOutMinimum` value is calculated based on the incorrect decimal.

```
function swapWithMaximumSlippage(
    address router,
    uint24 fee,
    address tokenIn,
    address tokenOut,
    address recipient,
    uint256 amountIn
) internal returns (uint256 amountOut) {
    uint256 amountOutMinimum = (amountIn * 970_000) / 1_000_000;
    amountOut = swap(
        router,
        fee,
        tokenIn,
        tokenOut,
        recipient,
        amountIn,
        amountOutMinimum
    );
}
```

Proof of Concept

An example of a test that simulates failing is shown below.

```
it ("Withdrawing other stablecoins is failed", async () => {
    const testAccount = "0x88a1493366D48225fc3cEFbdae9eBb23E323Ade3";
    const dai = new ethers.Contract(DAI, erc20_abi, ethers.provider);
    const vaultAddress = await dai_vault.getAddress();
    await hre.network.provider.request({
        method: "hardhat_impersonateAccount",
        params: [testAccount],
```

```
    });
    const signer = await hre.ethers.getSigner(testAccount);
    const TEN_DAI = ethers.parseUnits("10", 18);
    await dai.connect(signer).approve(vaultAddress, TEN_DAI);
    await dai_vault.connect(signer).deposit(TEN_DAI, testAccount);
    const shares = await dai_vault.balanceOf(testAccount);
    await dai_vault.connect(signer).queueWithdrawToken(shares, USDC);
    await mine(864000);
    // The following transaction is reverted with 'Too little received' error
    await dai_vault.connect(signer).withdraw(0);
});
```

Impact

Slippage protection is working incorrectly and the `queueWithdrawToken` function calls always fail if the input token's decimal is greater than the output token's decimal (e.g input token is DAI and output token is set to USDC).

Recommendation

Calculate the `amountOutMinimum` value based on the input and output tokens prices.

Alternatively, allow a minimum output amount inputted by users during the `queueWithdrawToken` call.

Status

Resolved

QA

[Q-01] SkyMoneyVault - Floating pragma

Description

The contract has `pragma solidity ^0.8.27` and it might allow the contracts to be deployed with a different version than the one used for testing.

Different pragma versions being used in test and mainnet may pose unidentified security issues.

Recommendation

Specify a specific version of Solidity in the pragma statement.

Status

Resolved

[Q-02] SkyMoneyVault - Unused imports

Description

The contract declares `IOracle`, `IDelegationManager`, and `ISwapRouter` imports but they are not used in the contract.

Recommendation

Remove unused imports.

Status

Resolved

[Q-03] SkyMoneyVault - Unused variables

Description

The contract declares the `pendingWithdraws` and `nonce` variables but they are not used in the contract. Having unused variables reduces the code quality.

Recommendation

Remove unused variables.

Status

Resolved

[Q-04] SkyMoneyVault#canWithdraw - Insufficient compatibility with specific use cases or interactions

Description

The `canWithdraw` function only allows the owner of the queue with input index to call it even if it's a view function.

If someone deposited tokens through external contracts and the contract doesn't have a function to interact with the `canWithdraw` function, then this function always returns false for the index of the queue which he created.

Recommendation

Remove the requirement to check whether the queue's owner is msg.sender.

Status

Resolved

Centralisation

The OpenZK 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 OpenZK 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