#Hashlock.

A51 Finance

(Concentrated Liquidity Tooling)

SMART CONTRACT

Security Audit

Performed on Project:

concentrated-liquidity-tool

Github Commit Hash:187fd382c7e7d8215a0e9dd39a81534ec7d3b238

Platform

EVM

hashlock.com.au

February 2024

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

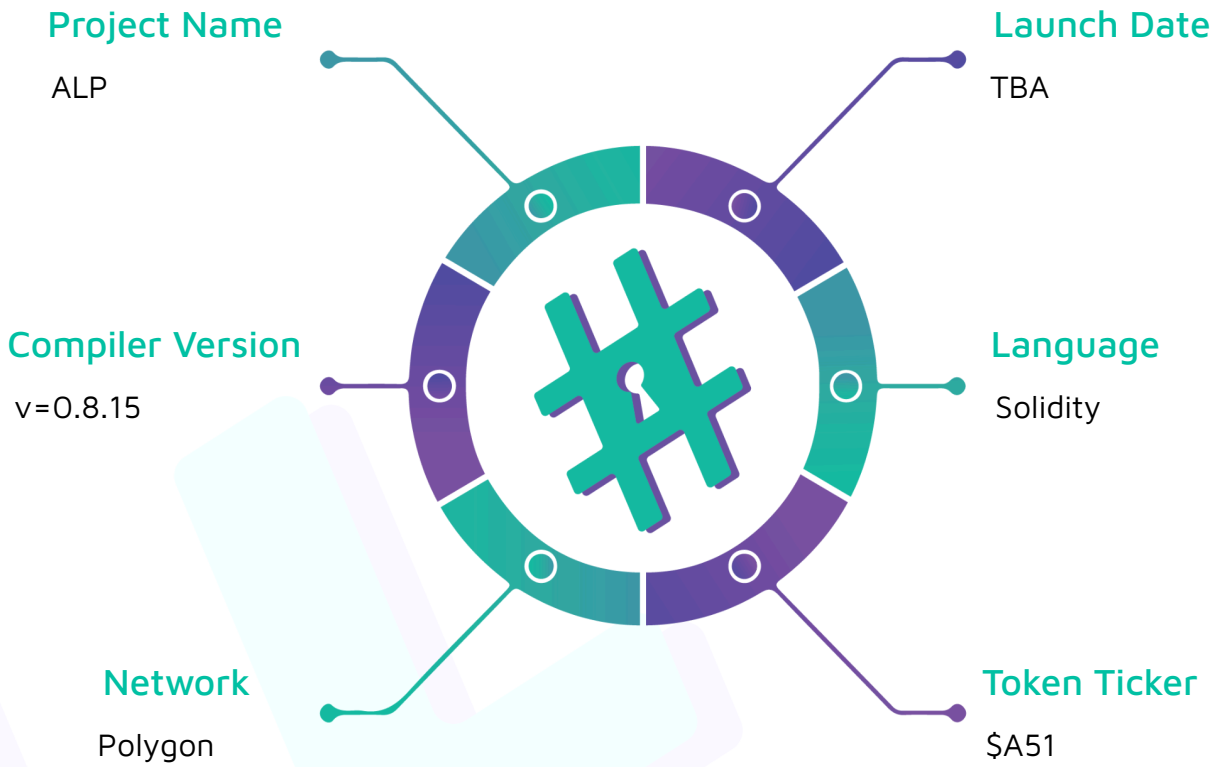
The A51 Finance team partnered with Hashlock to conduct a security audit of their Concentrated Liquidity smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

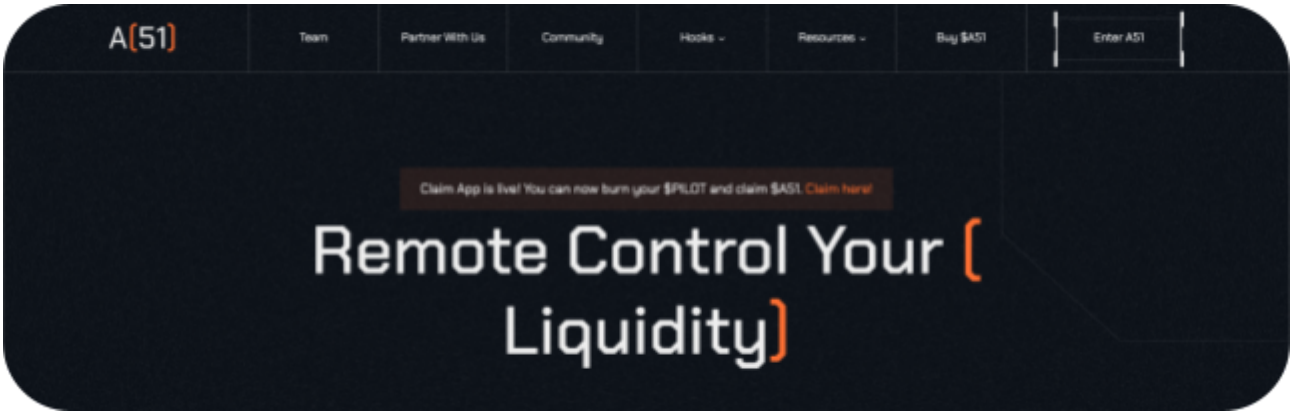
A51 Finance's Concentrated Liquidity Tool is a DeFi App built on the Polygon Network utilising QuickSwap which allows you to add tokens to a liquidity pool where various modules in the form of strategies will govern LP for maximum yield. These modules will manage the user's funds according to basic or advanced modes selected.

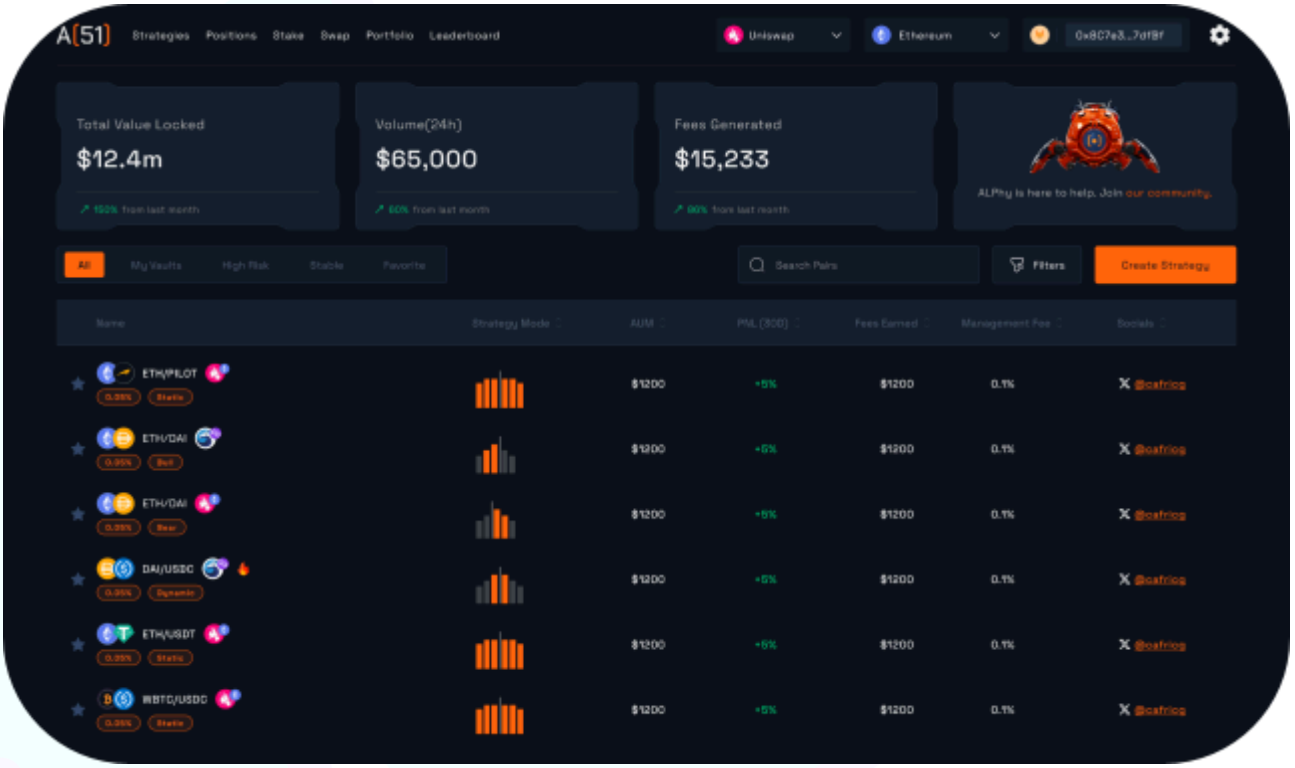
Project Name: ALP
Compiler Version: =0.8.15
Website: a51.finance
Logo:

Visualised Context:



Project Visuals:





Audit scope

We at Hashlock audited the solidity code within the Concentrated Liquidity Tooling project, the scope of works 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	**A51 ALP Protocol Smart Contracts**
Platform	**Polygon / Solidity**
Audit Date	**Feb, 2023**
Project 1	a51finance/concentrated-liquidity-tool
GitHub Commit Hash	187fd382c7e7d8215a0e9dd39a81534ec7d3b238

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 and Uniswap helper contracts. We initially identified some vulnerabilities which will need to be 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. 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 we have identified have yet to be resolved or acknowledged.

Hashlock found:

0 High severity vulnerabilities

4 Medium severity vulnerabilities

2 Low severity vulnerabilities

1 Gas Optimisations

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.*

Standardised Checks

Main Category	Subcategory	Result
General Code Checks	Solidity/compiler version stated	Passed
	Consistent pragma version across each contract	Passed
	Outdated Solidity Version	Passed
	Overflow/underflow	Passed
	Correct checks, effects, interaction order	Reviewed
	Lack of check on input parameters	Reviewed
	Function input parameters check bypass	Passed
	Correct Access control	Passed
	Built in emergency features	Reviewed
	Correct event logs	Passed
	Human/contract checks bypass	Passed
	Random number generation/use vulnerability	Passed
	Fallback function misuse	Passed
	Race condition	Passed
	Logical vulnerability	Reviewed
	Features claimed	Passed
	delegatecall() vulnerabilities	Passed
	Other programming issues	Reviewed
Code Specification	Correctly declared function visibility	Passed
	Correctly declared variable storage location	Passed
	Use keywords/functions to be deprecated	Passed
	Unused code	Passed
Gas Optimization	"Out of Gas" Issue	Passed

	High consumption ‘for/while’ loop	Passed
	High consumption ‘storage’ storage	Passed
	Assert() misuse	Passed
Tokenomics Risk	The maximum limit for mintage not set	Passed
	“Short Address” Attack	Passed
	“Double Spend” Attack	Passed

Initial Audit Result: Secure

Intended Smart Contract Functions

Contract	Claimed Behaviour
CLTBase.sol	User entrypoint into the protocol allowing them to create strategies, deposit, withdraw, shift liquidity and claim their position fees.
CLTModules.sol	Contains functionalities for managing different modes and their actions for strategies.
GovernanceFeeHandler.sol	Contains functionalities for permissioned users to manage governance fees when passed as parameters into strategies.
AccessControl.sol	Mostly made up of helper functions for accessibility to access control enforcement in contracts.
CLTPayments.sol	Made up of helper methods for safe token handling with custom logic.
ModeTicksCalculation.sol	Provides methods for computing ticks for basic modes of strategy.
Modes.sol	Provides functionalities to update ticks for basic mode of strategies.

Code Quality

This Audit scope involves the smart contracts of the Concentrated Liquidity Tooling, 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 is required.

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

Audit Resources

We were given the Concentrated Liquidity Tooling 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 understanding the overall architecture of the protocol.

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

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

No High Severity Bugs Were Discovered.

Medium

[M-01] CLTBase.sol#withdraw - Users will not be able to withdraw when the contract is paused

Description

The `CTLBase.sol` allows for admins to pause the contract to cease operations from the users however, users will not be able to withdraw their funds whilst the contract is paused.

Vulnerability Details

As it can be seen from the below function, the `withdraw` function uses the `whenNotPaused` where the function reverts if the contract is paused:

```
    /// @inheritdoc ICLTBase

    function withdraw(WithdrawParams calldata params)
        external
        override
        nonReentrancy
        whenNotPaused
        isAuthorizedForToken(params.tokenId)
        returns (uint256 amount0, uint256 amount1)
    {

    ...
```

Impact

User funds may be stuck in the contract for an arbitrary amount of time at the discretion of the contract administrator.

Recommendation

It's recommended that the `whenNotPaused` modifier is removed from the withdraw function.

Status

Resolved - Prescribed suggestions have been implemented.

[M-02] CLTBase.sol#updateStrategyBase - Users may pay more fees when withdrawing from the protocol

Description

The `CLTBase.updateStrategyBase` allows for the changing of strategy management fees and performance fees which happens immediately. Should the fees be increased to an unreasonable amount, users may have to pay double fees plus more when withdrawing from the protocol.

Vulnerability Details

As it can be seen from the withdraw function below, fees are charged to the users when tokens are removed from the protocol:

```
    /// @inheritdoc ICLTBase

    function withdraw(WithdrawParams calldata params)
        external
        override
        nonReentrancy
        whenNotPaused
        isAuthorizedForToken(params.tokenId)
        returns (uint256 amount0, uint256 amount1)
    {
        UserPositions.Data storage position = positions[params.tokenId];
        StrategyData storage strategy = strategies[position.strategyId];

    . . .
```

```
        (vars.balance0, vars.balance1) = transferFee(
            strategy.key,
           protocolFee.protcolFeeOnPerformance,
           strategy.performanceFee,
           vars.fee0,
           vars.fee1,
           owner,
           strategy.owner
         );


        vars.fee0 -= vars.balance0;
        vars.fee1 -= vars.balance1;


        (vars.balance0, vars.balance1) = transferFee(
            strategy.key,
            protocolFee.protcolFeeOnManagement,
            strategy.managementFee,
            amount0,
            amount1,
            owner,
            strategy.owner
        );
```

Impact

This may cause users to have to pay an unexpected amount when attempting to retrieve their tokens from the protocol.

Recommendation

It's recommended that there is a grace period for when performance and management fees become active. In addition to this, applying a maximum fee across the protocol is also necessary to prevent malicious strategy owners from charging the users more than what is fair.

Status

Maximum management fees **resolved**, but grace periods were not applied.

[M-03] PoolActions.sol#swapToken - Insufficient slippage protections when swapping tokens may cause sandwich attacks

Description

The `PoolActions` contract is used as a contract library to provide abstractions when performing operations against the uniswap pools. These actions include swapping tokens, burning and minting liquidity. Slippage protection is a key security mechanism built into Automatic Market Makers; however, the various functionalities in this contract do not employ these protections.

Vulnerability Details

Consider the `swapToken` function as an example below:

```
    function swapToken(
        IUniswapV3Pool pool,
        bool zeroForOne,
        int256 amountSpecified
     )
        external
        returns (int256 amount0, int256 amount1)
     {
        (uint160 sqrtPriceX96,,) = getSqrtRatioX96AndTick(pool);
        uint160 exactSqrtPriceImpact = (sqrtPriceX96 * (1e5 / 2)) / 1e6;
        uint160 sqrtPriceLimitX96 =
               zeroForOne ? sqrtPriceX96 - exactSqrtPriceImpact : sqrtPriceX96 +
exactSqrtPriceImpact;


         (amount0, amount1) = pool.swap(
            address(this),
            zeroForOne,
            amountSpecified,
            sqrtPriceLimitX96,
            abi.encode(ICLTPayments.SwapCallbackData({ token0: pool.token0(), token1:
```

```
pool.token1(), fee: pool.fee() }))
        );
    }
```

The function performs the swap correctly however, there is no check for the minimum amount of tokens expected out of the pool.

Impact

Since there is no check for minimum tokens out after the swap, users may receive an undesirable amount of tokens during market turbulence.

Recommendation

It's recommended that the various functionalities in the `PoolActions` contract check for `minTokensOut` against the amount of tokens received by the pool and revert if what the user received is less than expected.

Status

Resolved - The `sqrtPriceLimit` is now user supplied for the `swapTokens` which ensures that the price cannot be lower than this if the user is swapping zero for one.

TODO (Devs) : Natspec needs to be updated for `ICLTBase.ShiftLiquidityParams` to include the above changes.

[M-04] CLTBase.sol#createStrategy - Improper validation of msg.value results in unrefunded ETH which could be stolen

Description

The `createStrategy` function in the `CLTBase` contract allows for users to create various different strategies which governs how funds in LP Pools are managed. Should governance choose to do so, will be able to set a strategy creation fee however, msg.value is not validated against the `strategyCreationFeeAmount` obtained from `_getGovernanceFees`.

Vulnerability Details

This vulnerability occurs if a user accidentally overpays the strategy creation fee using the payable keyword in the `createStrategy` function. Below it can be seen that when a user creates a deposit transaction against the contract, the entire balance is transferred back to the `msg.sender` .

```
        if (address(this).balance > 0) {
            TransferHelper.safeTransferETH(_msgSender(), address(this).balance);
        }
```

Should a user accidentally overpay the strategy creation fee, the transaction can be backrun with a deposit call, stealing the excess ETH tokens.

Impact

Should the human error factor be met, ETH tokens may be stolen through the deposit function.

Recommendation

Increase the number of decimals of the `REWARDS_PRECISION` constant to `1e36`.

Status

Resolved - `refundETH` function now added at the end of the `createStrategy` .

Low/QA

[L-01] UserPositions.sol#updateUserPosition - Unsafe downcasting when determining self.tokensOwed0 and self.tokensOwed1

Description

The `updateUserPosition` function is used to collect up to a maximum amount of fees owed to a user from the strategy. `feeGrowthInside0LastX128` and `feeGrowthInside1LastX128` is passed into the function which are `uint256` values. These values are downcased to `uint128` which can result in integer truncation.

Recommendation

It's recommended that when values such as these are downcasted, the OpenZeppelin SafeCast library is used or that the contract checks that these values can fit into 128 bits. This was rated low in severity because of its unlikeliness to trigger.

Status

Resolved - Now using `safeCastExtended` .

[L/QA-02] CLTBase.sol#getUserFee - State updates should not happen in getters

Description

The `getUserFee` function attempts to update global variables each time this is called however, getters should not be used to update storage variables in case of accidental update.

Recommendation

It's recommended that global variables are updated through another mechanism dedicated to updating. Consider updating this function a view function which purely returns the maximum amount of fees owed to a specific user position.

Status

Acknowledged - The devs have commented that this is by design.

Gas

[G-01] CLTBase.sol#updatePositionLiquidity - Unnecessary writes and read operations on storage values

Description

In the `updatePositionLiquidity` function, there appear to be multiple operations performed on storage variables. SLOAD operations cost 100 gas units to read cold values which can be quite costly.

Recommendation

It's recommended that storage values are stored in memory and operated upon the value stored in memory. Once all operations are complete, the original storage variable is updated to that of the value stored in memory.

Status

Resolved.

Centralisation

To some extent, the Concentrated Liquidity Tooling project values security and utility over decentralisation.

The owner executable functions within the protocol increase security, functionality and allow for proper setup of the contracts but depend highly on internal team responsibility.

Conclusion

After Hashlocks analysis, the Concentrated Liquidity Tooling for A51 Finance seems to have a sound and well tested code base, however our findings must be resolved in order to achieve security. Overall, most of the code is correctly ordered and follows industry best practices. The code is well commented 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 are 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 whitebox 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, 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 contracts 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 are unlimited, the audit makes no statements or warranties on security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree 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 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.