#Hashlock.

Immersve Logic

SMART CONTRACT

Security Audit

Performed on Contracts:

FundsManagerLogic.sol
FundsStorageLogic.sol

Platform

EVM

hashlock.com.au

NOVEMBER 2023

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 Immersve team partnered with Hashlock to conduct a security audit of their protocol smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that it is ready for mainnet deployment.

Project Context

Immersve, as principal member of the Mastercard network, uniquely supports both centralised and decentralised payment experiences. Their technology enables users to spend digital cash, including cryptocurrencies, at any Mastercard-accepting merchant, whether online, in physical stores, or in the metaverse. Through the use of their smart contracts, Immersve ensures user control over funds while facilitating seamless transactions across various platforms.

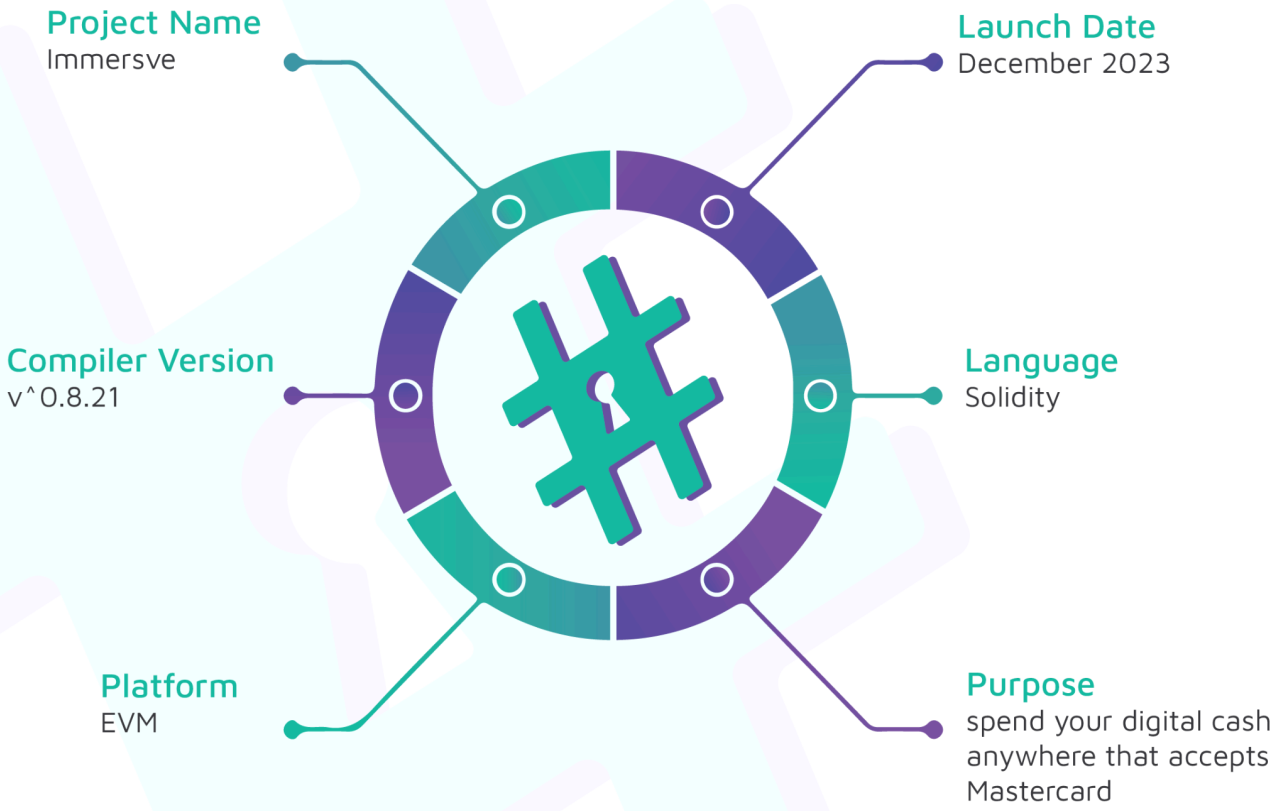
Project Name: Immersve
Contract Address: N/A
Compiler Version: ^0.8.21

Logo:

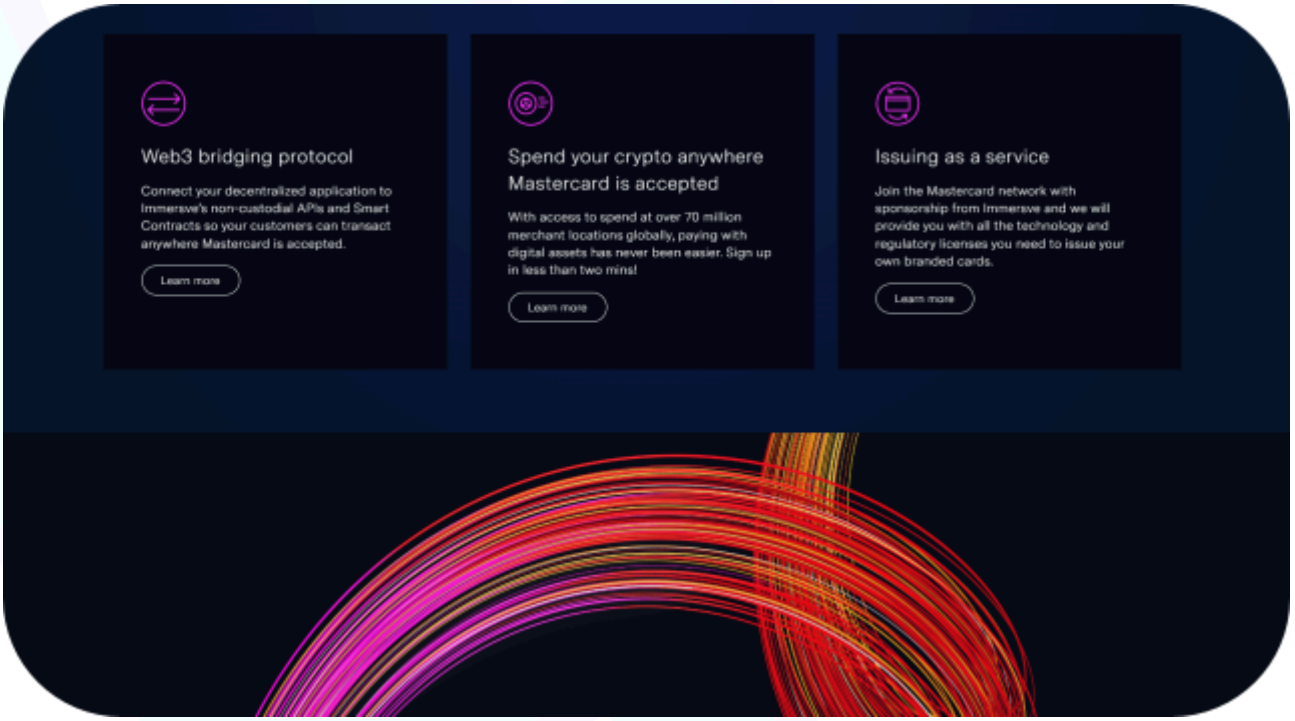
Immersve®

Be the master of your money

Visualised Context:



Project Visuals:



Audit scope

We at Hashlock audited the solidity code within the Immersve 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	**Project Review and Security Analysis Report for Immersve Smart Contracts.**
Platform	**Polygon PoS**
Audit Date	**November, 2023**
Commit	84ec9b53acc2fea1c75156c39ce33622cdc114e1
Contract 1	FundsManagerLogic.sol
Contract 1 SHA256	e434aebb0f2b5181035efc49c5060400584f995b9ad41884010bc6d721abca02
Contract 2	FundsStorageLogic.sol
Contract 2 SHA256	75f2045f192b828c09b088d095bf9232af6045168eccb097df2d1f212f360a94

Security Rating

After Hashlock's Audit, we now find 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 a few medium and low severity 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 overview section. General overview is presented in the Function list section and all identified issues can be found in the Audit Findings section.

All vulnerabilities initially identified have now been resolved.

Hashlock found:

2 Medium severity vulnerabilities

4 Low severity vulnerabilities

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.*

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	Passed
	Lack of check on input parameters	Passed
	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	Passed
	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: Failed

Revised Audit Result: Passed

Intended Smart Contract Functions

Claimed Behaviour	Actual Behaviour
Contract 1 FundsManagerLogic.sol Factory contract for deploying FundsStorage beacon proxy contracts. Handles settlements to Circle and user refunds	**Contract achieves this functionality.**
Contract 2 FundsStorageLogic.sol Contract for handling user deposits and withdrawals. Holds tokens.	**Contract achieves this functionality.**

Function List

Function list Legend:

- FundsManagerLogic.sol
- FundsStorageLogic.sol
- DeterministicDeployer.sol
- UupsPlaceholder.sol

	Function	Visibility/Modifiers	Conclusion
1	initialize(string calldata commitId, string calldata buildNumber) - FundsManagerLogic	public,onlyProxy	Reviewed
2	getStorageBeaconAddress()- FundsManagerLogic	public,view	No Issue
3	getStorageLogicAddress() - FundsManagerLogic	public,view	No Issue
4	getAdminAddress() - FundsManagerLogic	public,view	No Issue
5	getAdminLogicAddress() - FundsManagerLogic	public,view	No Issue
6	_authorizeUpgrade(address newImplementation) - FundsManagerLogic	internal,view,onlyRole (DEFAULT_ADMIN_ROLE)	No Issue
7	upgradeToAndCall(address newImplAddress, bytes memory data) - FundsManagerLogic	public,payable,onlyProxy	No Issue
8	getVersion() - FundsManagerLogic	external,pure	No Issue
9	getBuildNumber() - FundsManagerLogic	external,view	No Issue
10	createFundsStorage(address token, string calldata name) - FundsManagerLogic	external,onlyProxy	No Issue
11	isFundsStorage(address addr) - FundsManagerLogic	public,view	No Issue
12	settle(address from, uint256 amount, uint256 nonce, bytes32 merkleRoot) - FundsManagerLogic	external,onlyRole(SETTLER_ROLE),whenNotPaused	No Issue
13	refund(address refundee, uint256 amount, uint256 nonce) - FundsManagerLogic	external,onlyRole(SETTLER_ROLE),whenNotPaused	No Issue

14	requireTokenSupported(address token) - FundsManagerLogic	public,view	No Issue
15	getSettlementAddress(address token) - FundsManagerLogic	external,view	No Issue
16	transferDefaultAdminRole(address adminAccount) - FundsManagerLogic	external,onlyRole(DEFAULT_ADMIN_ROLE)	No Issue
17	grantRole(address) - FundsManagerLogic	public,pure	No Issue
18	grantRole(bytes32 /*role*/, address /*account*/) - FundsManagerLogic	public,pure,override(AccessControlUpgradeable, IAccessControl)	No Issue
19	grantWithdrawalSignerRole(address withdrawalSigner) - FundsManagerLogic	external,onlyRole(DEFAULT_ADMIN_ROLE)	No Issue
20	revokeWithdrawalSignerRole(address withdrawalSigner) - FundsManagerLogic	external,onlyRole(DEFAULT_ADMIN_ROLE)	No Issue
21	grantSettlerRole(address settlerAddress) - FundsManagerLogic	external,onlyRole(DEFAULT_ADMIN_ROLE)	No Issue
22	revokeSettlerRole(address settlerAddress) - FundsManagerLogic	external,onlyRole(DEFAULT_ADMIN_ROLE)	No Issue
23	grantRefundTargetManagerRole(address managerAddress) - FundsManagerLogic	external,onlyRole(DEFAULT_ADMIN_ROLE)	No Issue
24	revokeRefundTargetManagerRole(address managerAddress) - FundsManagerLogic	external,onlyRole(DEFAULT_ADMIN_ROLE)	No Issue
25	requireWithdrawalSignerAuthorized(address signerAuthorizer) - FundsManagerLogic	external,view,whenNotPaused	No Issue
26	setRefunderAddress(address refunderAddress)	external,onlyRole(DEFAULT_ADMIN_ROLE)	No Issue
27	getRefunderAddress()	external,view	No Issue

28	grantRefundTargetRole(address refundTarget)	external,onlyRole(REFUND_TARGET_MANAGER_ROLE)	No Issue
29	revokeRefundTargetRole(address refundee)	external,onlyRole(REFUND_TARGET_MANAGER_ROLE)	No Issue
30	pause()	external,onlyRole(DEFAULT_ADMIN_ROLE)	No Issue
31	unpause()	external,onlyRole(DEFAULT_ADMIN_ROLE)	No Issue
32	initialize(address adminAddress, address token, string calldata name) - FundsStorageLogic	public,initializer	No Issue
33	getName() - FundsStorageLogic	external,view	No Issue
34	getToken() - FundsStorageLogic	external,view	No Issue
35	getWithdrawalNonce(address depositor) - FundsStorageLogic	public,view	No Issue
36	withdraw(uint256 amount, uint256 expiryDate, uint256 nonce, bytes memory _signature) - FundsStorageLogic	external,nonReentrant	Reviewed
37	_verifySignature(bytes32 digest, bytes memory signature) - FundsStorageLogic	internal,view	No Issue
38	transferToSettlementAddress(uint256 amount) - FundsStorageLogic	external	No Issue
39	deploy(bytes memory bytecode, uint256 _salt) - DeterministicDeployer	external	No Issue

40	initialize(address owner) - UupsPlaceholder	public,initializer	No Issue
41	_authorizeUpgrade(address newImplementation)	internal,onlyRole(DEFAULT_ADMIN_ROLE)	No Issue
42	version()	external,pure	No Issue

Code Quality

This audit scope involves the solidity smart contracts of the Immersve project, as outlined in the Audit Scope section. All contracts, libraries and interfaces intend to follow standard best practices and to help avoid unnecessary complexity that increases the likelihood of exploitation.

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

Audit Resources

We were given the Immersve Protocol's smart contract code in the form of Github access.

As mentioned above, code parts are well commented. The logic is fairly straightforward, and therefore it is easy to quickly comprehend the programming flow as well as the complex code logic after some manual testing. 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 material loss of funds, asset loss, access denial, theft of value, and other critical issues that will result in the direct loss of funds controlled by the owners or the community.
Medium	Medium level vulnerabilities should be solved before deployment and require certain edge cases to be met in order for funds to be lost, stolen or inaccessible.
Low	Low level vulnerabilities are areas that lack best practices that may cause small complications in the future, but are less important than high and medium level vulnerabilities.
Gas	Gas Optimisations, issues and inefficiencies are addressed to save on gas fees when interacting with the protocol.

Audit Findings

High

No high severity vulnerabilities were discovered.

Medium

[M-01] FundsStorageLogic#withdraw - Missing receiver address parameter causing USDC to be stuck in the contract

Description

The `withdraw` function sends tokens to `msg.sender` instead of allowing the caller to specify who to send the tokens to. This may lead to frozen tokens in the case if `msg.sender` cannot receive the tokens (such as in cases where the user is blacklisted).

Vulnerability Details

The `withdraw` function allows a user to withdraw tokens back to the `msg.sender` calling the function. If the user is unable to receive the tokens (such as when the user may be blacklisted in the case of USDC), there is no way to rescue those funds from the contract and those tokens will be lost forever.

Impact

If a user cannot receive tokens since they're blacklisted, these tokens would be frozen.

Recommendation

Locked funds can be handled in two ways:

1. Create a `rescueTokens` function to be used by the admin of the contract in order to pull those funds out. This would require a change in the `FundsStorageLogic` contract.
2. Handle frozen tokens in the Immersve backend by transferring the blacklisted user's Immersve balance to an admin-owned account specifically for locked funds. The admin-owned account can then go through the standard withdrawal process to retrieve the locked funds.

If the funds are blacklisted USDC, the `isBlacklisted` function inside the USDC contract can be used to check if a user is blacklisted. After retrieving the blacklisted funds, perform the usual necessary steps to comply with laws and regulations.

Status

Resolved using external technologies

[M-02] Lack of `safeTransfer` when attempting to move funds which can cause tokens to behave unexpectedly

Description

ERC-20 transfers do not use `safeTransfer` from OpenZeppelin's `SafeERC20` library. This could lead to unintended behavior.

Vulnerability Details

The ERC-20 standard states that ERC-20 tokens should `throw` if the message caller's account balance does not have enough tokens to send. This means that on unsuccessful transfers, the transaction should revert.

However, there are ERC-20 tokens that exist that do not revert on unsuccessful transfers and return `false` on unsuccessful transfers. In these cases, execution will not be reverted and Immersve's smart contracts will behave as if the transfer was successful.

Impact

In the case that one of these tokens are being used in Immersve's smart contracts, the smart contracts and Immersve's backend will treat token transfers as being successful even if they have failed. This can lead to unintentional behavior such as user funds being deducted and lost, or Immersve adding user balances when the user has unsuccessfully transferred tokens.

Recommendation

Use the `safeTransfer` and `safeTransferFrom` functions inside OpenZeppelin's `SafeERC20` library.

Status

Resolved

Low

[L-01] Interface files should be prefixed with `I`

Description

It is standard practice in Solidity to prefix interface files with `I`. This allows for better code readability.

Recommendation

Prefix all interface files with `I`.

Status

Resolved

[L-02] FundsManagerLogic#initialize - Function can be called by anyone

Description

Since there is no access control apart from `onlyProxy` and there is no `initializer` modifier, the `initialize` function can be called by any address for any number of times from the proxy, as long as `_proxiesInitialized = true`.

This means that any address can change the `_buildNumber` and `_commitId` state variables.

Recommendation

Instead of using `_initialize` which is protected by the `reinitializer()` modifier, use the `initializer` modifier in `initialize`, such that the `_fundsStorageBeacon` state variable can only be assigned once.

Instead of setting the `_buildNumber` and `_commitId` inside `initialize`, this can be done in its own access-controlled setter functions.

Status

Resolved

[L-03] FundsManagerLogic - Missing merkle root implementation

Description

The project documentation and component diagrams specify that there are merkle root updates whenever `settle` or `refund` is being called. This functionality has not been implemented.

`settle` takes in a `merkleRoot` parameter but does not store it or perform any logic with it.

`refund` does not take in a `merkleRoot` parameter

Recommendation

Store merkle root updates in `settle` and `refund`.

Status

Resolved

[L-04] FundsManager#settle - Missing `from` sanity check

Description

The `settle` function does not check whether the `from` input parameter is a `FundsStorage` contract.

Recommendation

Add a check to make sure that `from` is a `FundsStorage` contract using the `isFundsStorage` function.

Status

Resolved

Gas

[G-01] FundsManagerLogic#refund - Avoid doubling up on external calls by caching to memory

Description

The `refund` function calls `fundsStorage.getToken()` twice.

```
requireTokenSupported(fundsStorage.getToken());
IERC20 token = IERC20(fundsStorage.getToken());
```

Recommendation

To save on gas spent on external calls, do the external call once and cache it into a memory variable, then reuse that memory variable.

Status

Resolved

Centralisation

Immersve prioritises security and utility, skillfully blending the benefits of decentralisation with the global reach of the centralised Mastercard network. This innovative approach allows users to conduct digital currency transactions at any Mastercard-accepting location worldwide, maintaining elements of decentralisation while ensuring a secure and versatile payment experience.

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

Conclusion

After Hashlocks analysis, the Immersve project seems to have a sound and well tested code base, now that our 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.

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