#Hashlock.

Algem (Staking)

SMART CONTRACT

Security Audit

Performed on Contracts:

ALGMStaking.sol
MD5 Hash:f18c2fe1b45226f8dc3f8cbae0cd0d47
veALGM.sol
MD5 Hash:d4b2caa5c29abc8f0dda0c7ab68b27f3
Github Commit Hash:2e7d43343b1be4dde3b7388a2965218babd739a9

Platform

ASTR

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 Algem team partnered with Hashlock to conduct a security audit of their Governance Staking 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

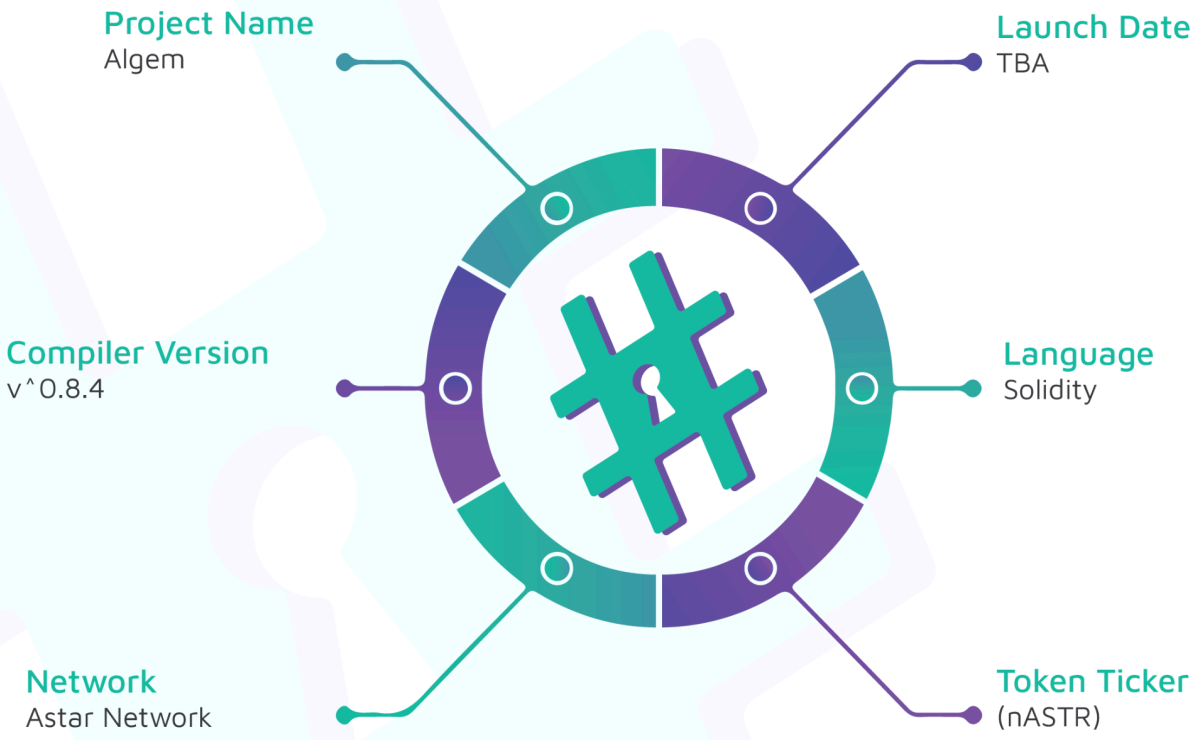
Algem is a DeFi dApp built on Astar Network that allows you to stay liquid while staking your ASTR. Staying liquid means you can double-dip with your Astar tokens by staking while yield farming.

Simply put, you don't have to choose between staking and yield farming with your Astar tokens. You can do both.

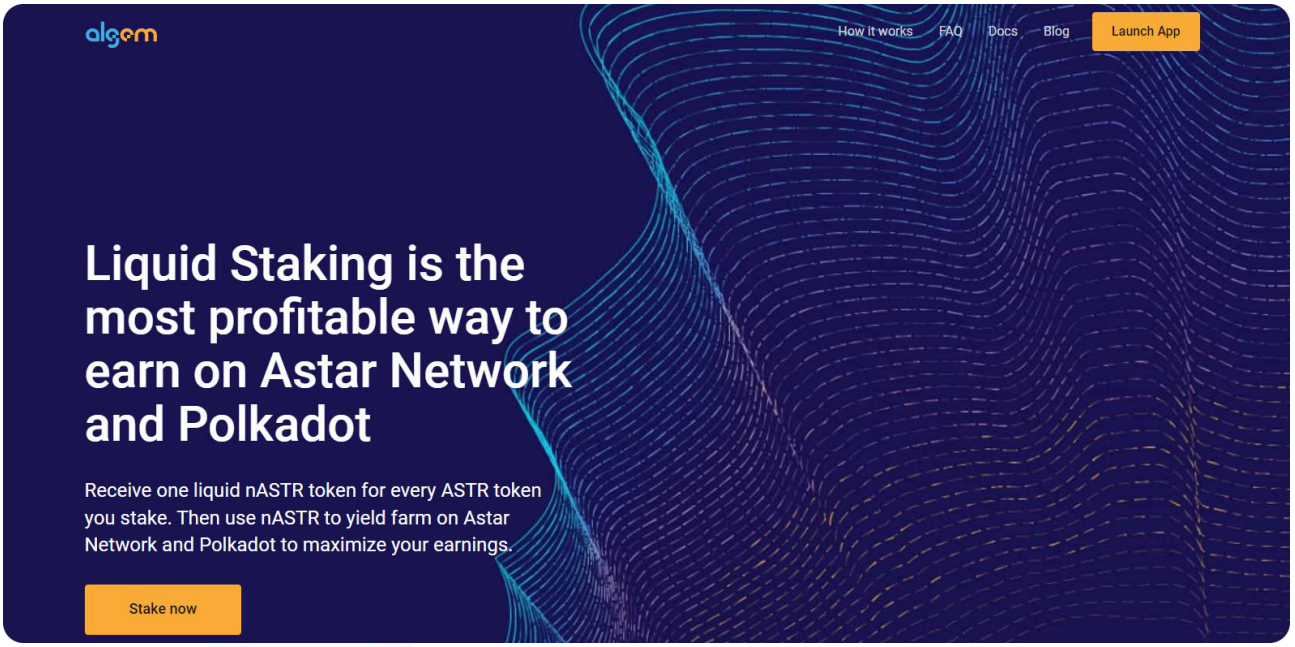
Project Name: Algem
Compiler Version: ^0.8.4
Website: https://www.algem.io/

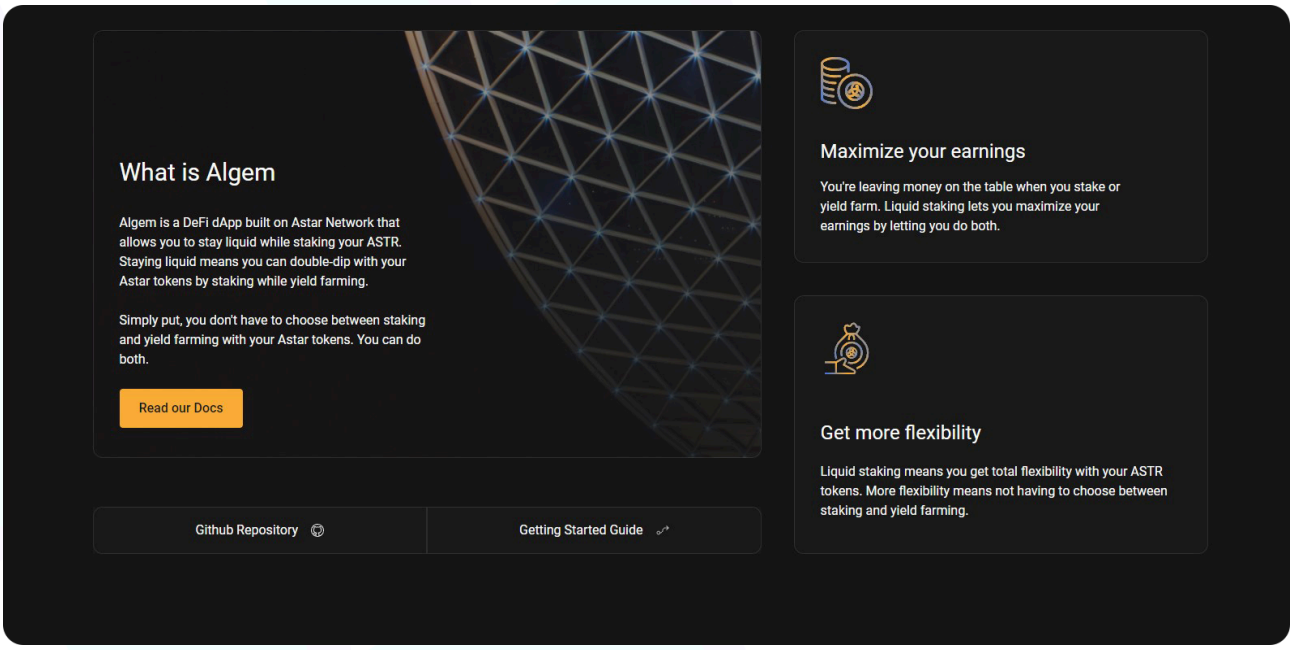
Logo:

Visualised Context:



Project Visuals:





Audit scope

We at Hashlock audited the solidity code within the Algems Staking 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	**Project Review and Security Analysis Report for Algem Protocol Smart Contracts and other factors.**
Platform	**Ethereum / Solidity**
Audit Date	**August, 2023**
Contract 1	ALGMStaking.sol
Contract 1 MD5 Hash	f18c2fe1b45226f8dc3f8cbae0cd0d47
Contract 2	veALGM.sol
Contract 2 MD5 Hash	d4b2caa5c29abc8f0dda0c7ab68b27f3

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 overview section. General overview is presented in the Function list section and all identified issues can be found in the Audit overview section.

All vulnerabilities initially identified have now been resolved and acknowledged.

Hashlock found:

2 High severity vulnerabilities

5 Medium severity vulnerabilities

2 Low severity vulnerabilities

2 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	Reviewed
	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	Reviewed
	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	Reviewed
Gas Optimization	"Out of Gas" Issue	Reviewed
	High consumption 'for/while' loop	Reviewed

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

Initial Audit Result: VULNERABLE

Revised Audit Result: PASSED

Intended Smart Contract Functions

Claimed Behaviour	Actual Behaviour
ALGMStaking - Allows users to stake and unstake their ALGM tokens in exchange for rewards. - Allows admins to have access to special functions for the purpose of configuration and settings up rewards tokens.	The contract achieves this behaviour however, as it can be seen from the below findings, there may be some instances where rewards tokens may be stolen or denial of service conditions may be met.
veAlgm - Acts as an ERC20 token contract which is given to users as a representation of their staking position in the staking contract.	This contract achieves its intended behaviour.

Code Quality

This audit scope involves the solidity smart contracts of the Algem 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 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 Algem Protocol's 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

[H-01] Malicious users can deposit after bonding period to steal rewards from other stakers

Description

The ALGMStaking contract allows for users to stake their ALGM tokens and claim rewards in the form of partner or ASTR tokens. These rewards accumulate over a period of time to which users can unstake their tokens and claim their rewards; however, user rewards will be inflated should they stake a significant amount after a certain period of time.

Vulnerability Details

When the contract calculates the quantity of ALGM staked into the contract, this value can be added to or removed from at any point during the staking period and the contract will believe that users have been staking that value from the beginning. This will give exploiters an opportunity to inflate their entitled rewards by staking a significant amount at a later date.

Impact

Theft of rewards from other users in the same pool. The proof of concept below outlines this scenario:

```
function testTheftOfStakingRewards() public {
    // Setup scenario
    algm.mint(address(eve), 30_000 ether); // issue initial balance to the user
    algm.mint(address(alice), 10_000 ether);
    algm.mint(address(bob), 10_000 ether);

    vm.startPrank(alice);
    algm.approve(address(staking), type(uint256).max);
    staking.stake(0, 10_000 ether);
    vm.stopPrank();

    vm.startPrank(bob);
    algm.approve(address(staking), type(uint256).max);
    staking.stake(0, 10_000 ether);
    vm.stopPrank();

    // Malicious user deposits the same amount into the staking contract using a
malicious contract
```

```
    vm.startPrank(eve);
    algm.approve(address(staking), type(uint256).max);
    staking.stake(0, 10_000 ether);
    vm.stopPrank();

    vm.warp(10 days); // set block timestamp to accumulate rewards

    // 10 days have passed and it's time to collect rewards
    vm.startPrank(address(eve));
    // Claimable rewards before

    (
      uint256 astrQty,
      IERC20[] memory tokensList,
      uint256[] memory tokensQty,
      bool rewardsAvailableFlag
    ) = staking.calculateRewards(address(eve), 0);

    uint256 astrQtyBefore = astrQty;

    staking.stake(0, 20_000 ether);

    (
       astrQty,
       tokensList,
        tokensQty,
       rewardsAvailableFlag
    ) = staking.calculateRewards(address(eve), 0);

    uint256 astrQtyAfter = astrQty;

        vm.stopPrank();
        // Asserts that staking rewards are immediately inflated
    assertGt(astrQtyAfter, astrQtyBefore);

}
```

Recommendation

It's recommended that user stakes are separated out into different buckets for example stakes[poolID][staker][stakeID] which will allow each staking transaction to be handled separately when calculating rewards. An alternative solution is to deny users from staking if they already have ALGM tokens in the contract and when they withdraw, they are forced to withdraw their entire stake.

Hacker's notes

- Possibly susceptible to flashloan attacks

Status
Invalid - Waiting 10 days does not generate rewards. 10 days is just an un-bonding period during which one can not withdraw their staked ALGM.

[H-02] rpTokensQty[token] is not updated in topUpRewardsPoolFor which will cause a dos when attempting to claim rewards

Description

The topUpRewardsPoolFor function will allow an owner to add rewards to certain ASTR or partner tokens. Users can claim these rewards for staking their ALGM tokens in the contract. msg.value is accounted for if ASTR tokens are used and ERC20 is used for transferring other tokens to the contract.

Vulnerability Details

Various checks are made on the quantity and token being used to top the rewards pool up however, rpTokensQty is not updated with the amount of qty which has been added to the pool which effectively means, a top up never happened. Because this state variable is heavily relied upon when calling the claimRewards function, this may cause an underflow which will deny users from claiming their rewards.

Impact

This will create a denial of service condition against the claimRewards function.

Recommendation

It's recommended that the rpTokensQty state variable is updated when an owner tops up the reward pool in topUpRewardsPoolFor function.

Status
Resolved

Medium

[M-01] If a partner reward token if removed users will lose rewards

Description

The staking contract owner can add partner tokens to the protocol in order to administer rewards in this type of token through the calculation of rewards. If an owner deletes a partner token where there are pending rewards, the users may lose rewards for their staking efforts.

Vulnerability Details

This is due to the lack of checks when attempting to delete a partner reward token. If a token is removed, this is not accounted for in the claimRewards and _calculateRewards function which effectively loses the accumulated rewards of users.

Impact

Users will lose their rewards if a partner reward token is deleted.

Recommendations

Check if there are pending rewards before owners remove partner tokens.

Status

Resolved

[M-02] Time distributed reward updates are sometimes skipped

Description

The _updateTimeDistRewards function is responsible for updating the ARPS for tokens which are distributed within TimeDistRewards. This will then delete the completed TimeDistRewards entry from the storage of the contract; however, time distributed rewards may be skipped.

Vulnerability Details

Because this deletes the TDR by replacing the last element with the current one then popping the last element within a for loop, the last TDR will be missed since it's being moved to the current index which has already been processed.

Impact

User rewards may be skipped

Recommendations

It's recommended that there are some additional checks on the token quantity in addition to the timeframe and slicesDistributed to ensure that time dist reward indexes aren't accidentally deleted.

Hacker's notes

- Discovered in ALGMStaking#_updateTimeDistRewards

Status

Resolved

[M-03] Stake's veAlgmQty is not reset if user unstakes all ALGM

Description

If the user unstakes their entire staked algmQty, then the veAlgmQty value inside their stake does not get reset back to 0.

Vulnerability Details

If a user unstakes all of their ALGM tokens, then the veAlgmQty does not change if stakedQty == unstakeQty in the unstake function. In the future once guages are implemented, this may be exploited to have greater voting power than intended.

Impact

The user's stake.veAlgmQty value will show that they still have veALGM even though it has been burnt. If this value is to be used for gauge weight voting in the future, it is possible for an exploiter to be able to vote with veALGM that they no longer hold.

Recommendations

It's recommended that the quantity of veAlgm is updated to zero if they unstake all of their tokens.

Hacker's notes

- Discovered in ALGMStaking#unstake

Status

Resolved

[M-04] ALGMStaking - Precision loss due to division before multiplication

Description

There are multiple instances in ALGMStaking where division is applied before multiplication. In particular, _updateTokenARPSinPools and _calculateVeALGM both divide before multiplying. This results in lost rewards and veALGM.

Vulnerability Details

Because Solidity handles multiplication before division differently to division before multiplication due to floating point errors, we instead get rounding errors. By doing the multiplication operations first, we can mitigate the rounding related issues as much as possible.

Impact

Precision loss when attempting to calculate veAlgm tokens. This impact can be summed up in the following example:

```
console.log((30 * 100 * 13) / 13)
> 3000

console.log((30 / 13) * 100 * 13)
> 2999.9999999999995
```

Recommendations

It's recommended that multiplication operations are done before division operations throughout the code base.

Status

Resolved

[M-05] If WASTR is used as a partner token the claimRewards function could be dosed or double rewards may be claimed

Description

The claimRewards function is being used to reward users for their staking efforts through both ASTR tokens and partner tokens. Both instances of rewards are iterated through with a for loop with minimal checks which may cause some issues if WASTR is accidentally added as a partner token.

Vulnerability Description

Because the claimRewards function does two for loops with minimal checks, when WASTR token is passed to rpTokensQty when attempting to determine the ERC20 rewards, the WASTR address could be used twice. If the rewards pool for WASTR was not funded this could create a dos however if the rewards pool was funded with WASTR, this could create duplicate rewards as ASTR token was transferred to the user via a call() in the previous for loop iteration.

Impact

This could result in a denial of service condition or double claim in rewards.

Recommendations

Because the WASTR address is already considered in the state variables, it's recommended that owners are not allowed to add WASTR as a partner token. In addition to this, it's recommended that sanity checks are implemented to prevent

WASTR from being transferred as an ERC20 when it was already transferred in a low level call as native tokens in the previous iteration.

Status

Resolved

Low

[L-01] Missing sanity checks for instances of msg.value

Recommendations

Ensure that msg.value is zero so tokens are not lost.

Status

Resolved

Hacker's Notes

- Discovered in ALGMStaking#topUpRewardsPool

[L-02] to can be set to zero address

Recommendations

If from != address(0) then to can be the zero address. It's recommended that this edge case is checked for.

Hacker's notes

- Discovered in ALGMStaking/veALGM#_revertIfInvalidAddr

Status

Resolved

Gas

[G-01] Initialize calcPredicted as false

Hacker's notes

- Discovered in ALGMStaking#_calculateRewards

Status

Resolved

[G-02] Storing totalPartnerTokens is unnecessary as partnerTokens is already stored

Hacker's notes

- Discovered in ALGMStaking

Status

Resolved

Centralisation

The 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 Algem project seems to have a sound and well tested code base, however our findings need to 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.