hashlock.

2024

Security Audit

Algem Liquid Staking (DeFi)

December 2024

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

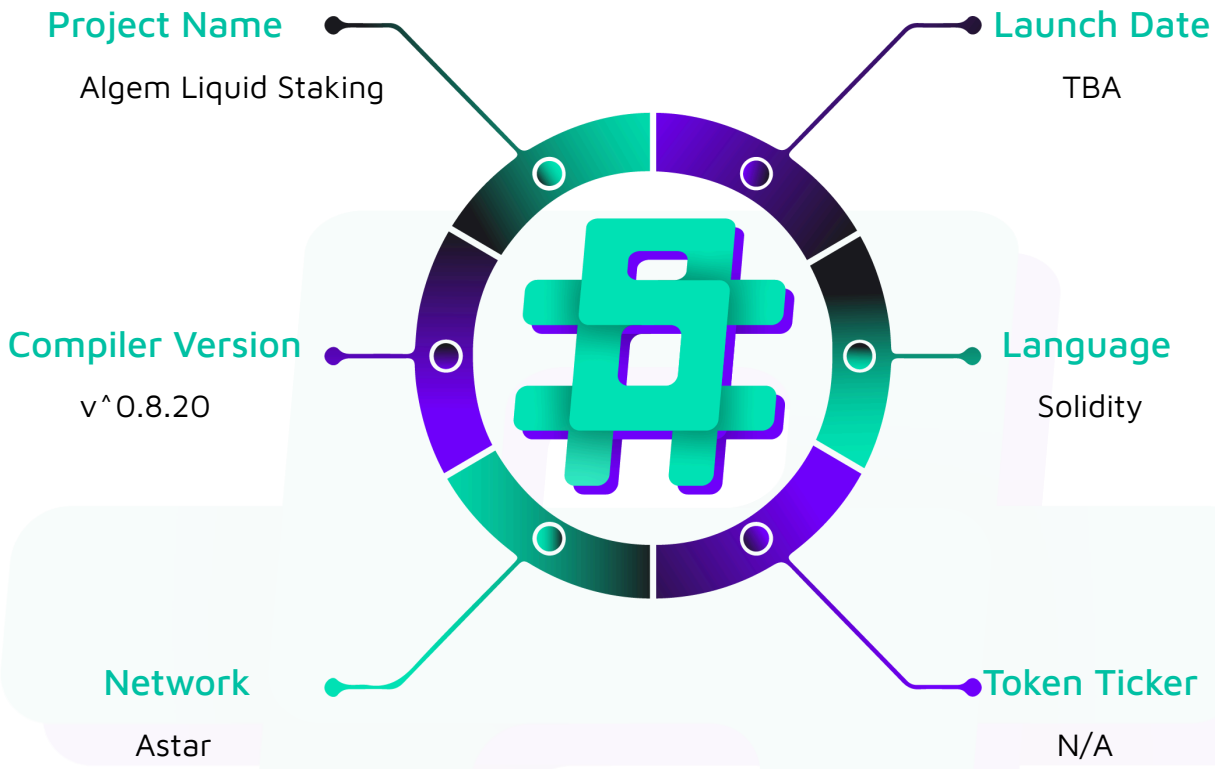
The Algem Liquid Staking team partnered with Hashlock to conduct a security audit of their LiquidStakingLayer2.sol, WASTRCCT.sol, LiquidStakingMain.sol and other smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

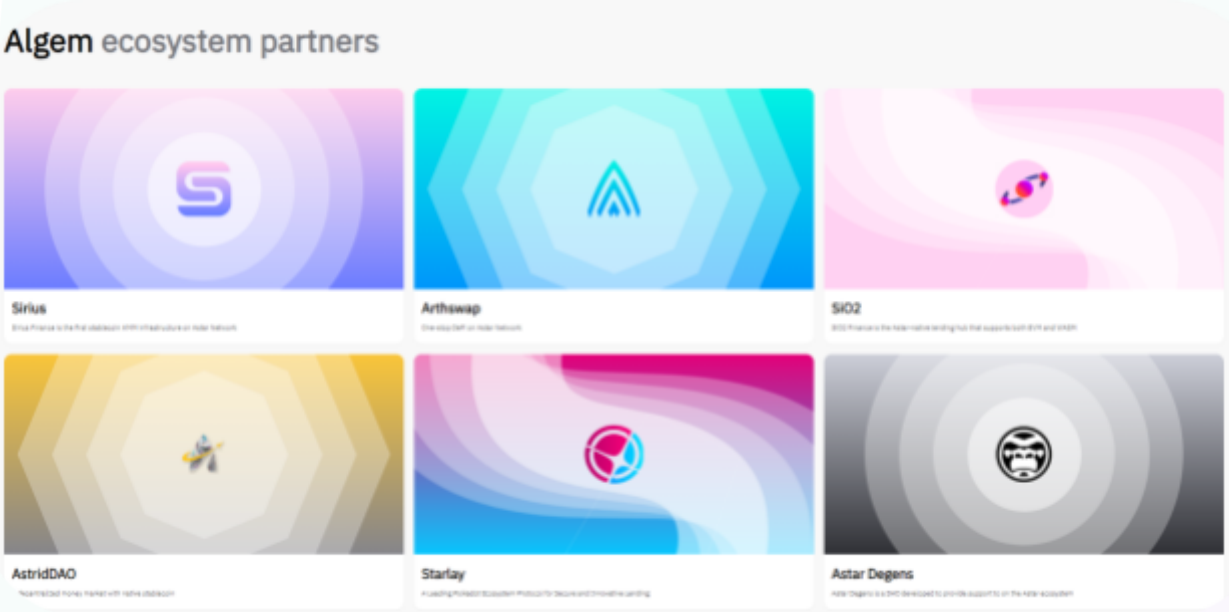
Algem.io is a decentralized liquid staking platform built on the Polkadot ecosystem, allowing users to stake assets while maintaining liquidity through Web3 technologies. It enables participants to earn staking rewards without locking up their tokens, offering flexibility to engage in DeFi activities like lending and yield farming. By providing non-custodial, liquid staking, Algem empowers users to maximize capital efficiency and compound their returns, positioning itself as a key player in the decentralized finance space within the Polkadot network.

Project Name: Algem Liquid Staking
Compiler Version: ^0.8.20
Website: www.algem.io
Logo:

Visualised Context:



Project Visuals:



Audit scope

We at Hashlock audited the solidity code within the Algem Liquid Staking 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	Algem Liquid Staking Protocol Smart Contracts
Platform	**Solidity**
Audit Date	**December, 2024**
Contract 1	LiquidStakingLayer2.sol
Contract 1 MD5 Hash	fd710f66e80ebe8e68fec9ee728aa9a8
Contract 2	WASTRCCT.sol
Contract 2 MD5 Hash	792602310d7e2900feb6253e43507578
Contract 3	XNASTR.sol
Contract 3 MD5 Hash	82a5fb878e1cedaa1622d24a57c3e4fd
Contract 4	LiquidStakingAdmin.sol
Contract 4 MD5 Hash	4495fda92c69235768363c8e803757e3
Contract 5	LiquidStakingMain.sol
Contract 5 MD5 Hash	e6bc164a73bc5327abf8d57a4ef49404
Contract 6	LiquidStakingManager.sol
Contract 6 MD5 Hash	be305ff1176b9cd8783cb37ee8c836a6
Contract 7	LiquidStaking.sol
Contract 7 MD5 Hash	15716686d76b78f31df7fa47bd7adf73
Contract 8	LiquidStakingStorage.sol
Contract 8 MD5 Hash	2d623cbf365537f7fa298b9ecf464ddd
Contract 9	LiquidStakingVoting.sol

Contract 9 MD5 Hash	dc83c85d76dbb078c9791af9e3f71482
Contract 10	BaseOFTV2Upgradeable.sol
Contract 10 MD5 Hash	42cf1da5e75ee3ca156b911a5c3980b9

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:

1 Medium severity vulnerabilities

4 Low severity vulnerabilities

3 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
LiquidStakingLayer2.sol - A Chainlink CCIP-enabled Layer 2 contract that handles liquid staking of wASTR tokens and manages voting functionality using veALGM tokens. - Enables users to stake wASTR to receive xnASTR tokens, perform unstaking operations, and participate in dapp voting with built-in cross-chain messaging.	**Contract achieves this functionality.**
WASTRCCT.sol - A wrapper token contract for ASTR that provides wrapped token functionality with mint/burn controls and CCIP integration.	**Contract achieves this functionality.**
XNASTR.sol - An ERC20 token contract (XNASTR) representing staked ASTR positions with mint/burn role management.	**Contract achieves this functionality.**
LiquidStakingAdmin.sol - An admin contract for managing liquid staking system parameters, including NFT discounts, stake amounts, and dapp configurations.	**Contract achieves this functionality.**
LiquidStakingMain.sol - Handles staking ASTR tokens, distributes rewards, manages NFT-based cashback, and implements sophisticated weight calculations for	**Contract achieves this functionality.**

dapp distribution.	
LiquidStakingManager.sol - Enables admin-controlled addition, deletion, and modification of selector-to-address mappings with proper access controls.	**Contract achieves this functionality.**
LiquidStaking.sol - A core liquid staking contract that acts as a proxy and handles cross-chain communication through Chainlink CCIP for stake/unstake operations. - Manages ASTR/wASTR conversions, handles CCIP message processing for staking operations, and implements pause functionality with granular control per function.	**Contract achieves this functionality.**
LiquidStakingStorage.sol - Provides state variables for Liquid Staking	**Contract achieves this functionality.**
LiquidStakingVoting.sol - A voting contract that manages dapp registration and handles vote weight distribution for liquid staking rewards. - Enables adding/removing dapps, setting their weights, and updating user vote balances with internal access controls.	**Contract achieves this functionality.**
BaseOFTV2Upgradeable.sol - OFTV2 Token implementation	**Contract achieves this functionality.**

Code Quality

This audit scope involves the smart contracts of the Algem Liquid Staking 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 Algem Liquid Staking 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

Medium

[M-01] LiquidStakingMain.sol - Potential NFT Lock Loss

Description

The `addCashbackLock` and `releaseCashbackLock` functions contain a design flaw in NFT ID management. When a user's lock amount is 0, depositing a second NFT from the same contract overwrites the first NFT's ID, making it irretrievable.

Vulnerability Details

The `_ccipSend` function in line 283:

```
function addCashbackLock(address _nftAddr, uint256 _amount, uint256 _tokenId) external updateAll {
    // ...
    CashbackLock storage lock = cashbackLocks[msg.sender][_nftAddr];
    if (lock.amount == 0) {  // Only occurs when lock amount is 0
        if (nftContract.balanceOf(msg.sender) == 0) revert NotEnoughNFTForLock();
        lock.tokenId = _tokenId;  // Overwrites any previous tokenId
        nftContract.transferFrom(msg.sender, address(this), _tokenId);
    }
    // ...
}

function releaseCashbackLock(address _nftAddr, uint256 _amount) external updateAll {
    // ...
    CashbackLock storage lock = cashbackLocks[msg.sender][_nftAddr];
```

```
    if (!nft.isActive) revert WrongNFTRelease();

    if (lock.amount == 0) revert NoCashbackLocks();

    // Can only withdraw the most recently stored tokenId

    // ...

}
```

The issue occurs because the mapping `cashbackLocks[msg.sender][_nftAddr]` only store one NFT ID per user per contract. When `lock.amount` is 0 and a user deposits multiple NFTs with different IDs from the same contract, the second deposit overwrites the first NFT's ID, making the first NFT permanently locked.

Impact

While being an edge case, users can lose their NFT which will become locked in the contract.

Recommendation

Disallow 0 lock amounts. Track not only the contract address but also the ID.

Status

Resolved

Low

[L-01] LiquidStakingManager - Single address granted multiple privileged roles creates centralization risk

Description

In `LiquidStakingManager.sol`, the `initialize()` function grants both `DEFAULT_ADMIN_ROLE` and `MANAGER` roles to the same address (msg.sender). This concentration of privileged roles in a single account creates a single point of failure and increases the risk if the account is compromised.

Additionally, this setup goes against the principle of separation of duties, which is important for system security and management.

Recommendation

Consider separating the admin and manager roles to different addresses during initialization.

Status

Acknowledged

[L-02] Multiple contracts - Cross-chain message handler lacks error handling for invalid methods

Description

In both `LiquidStakingLayer2.sol` and `LiquidStaking.sol`, the `_ccipReceive` function uses a series of if statements to handle different cross-chain methods but lacks any error handling for cases where the method signature is malformed or doesn't match any of the expected methods.

This silent failure mode could lead to lost messages and make debugging cross-chain interactions more difficult. When a message arrives with an unexpected method signature, the function will complete execution without any indication of failure.

Recommendation

Implement explicit error handling for unrecognized methods. This could be done by adding a final else clause that either reverts with a custom error or emits an event to log the failed message.

Additionally, consider using a revert statement with a custom error like InvalidCCIPMethod to ensure unhandled methods are properly caught and reported.

Status

Resolved

[L-03] Multiple contracts - Use of transfer/transferFrom instead of safe alternatives reduces maintainability

Description

Throughout the project, token transfers are implemented using `transfer()` and `transferFrom()` instead of their safer alternatives `safeTransfer()` and `safeTransferFrom()`. While this is currently safe as the project interacts with known tokens, it creates maintenance overhead for future token integrations and doesn't follow current best practices.

The OpenZeppelin safe versions handle non-standard token implementations better and provide additional safety checks.

Recommendation

Replace all instances of `transfer()` and `transferFrom()` with `safeTransfer()` and `safeTransferFrom()` from OpenZeppelin's SafeERC20 library. This change will improve code maintainability and make future token integrations more robust.

Status

Resolved

[L-04] LiquidStakingAdmin - Inefficient NFT management

Description

The LiquidStakingAdmin contract's `addNft` function uses a simple array (nftList) to store NFT addresses without duplicate checking or removal capability.

This implementation has two issues:

- First, the same NFT address can be added multiple times since there's no duplicate validation.
- Second, there's no mechanism to remove NFTs from the list, which limits contract maintainability.

Recommendation

Replace the current array implementation with OpenZeppelin's EnumerableSet library for NFT address storage.

This would provide built-in duplicate prevention and efficient removal capabilities. Additionally, implement a `removeNft` function to allow authorized users to remove NFTs when needed.

Status

Resolved

QA

[Q-01] Multiple contracts - Events not emitted on key state changes

Description

Several functions that modify key state variables do not emit events. This is observed in LiquidStaking.sol for functions such as `pause()`, `unpause()`, `setLiquidStakingManager()`, and `setPauseOnFunc()`.

While this might be a design choice, emitting events for significant state changes is considered best practice as it aids in contract monitoring, off-chain tracking, and integration testing. Events serve as an on-chain audit trail for important contract changes.

Recommendation

Consider emitting events for all functions that change key contract states to improve contract monitoring and transparency.

Status

Resolved

[Q-02] LiquidStakingMain - Inconsistent naming convention for private and internal functions

Description

In `LiquidStakingMain.sol`, the private function `calcWithWeights` and internal function `updates(uint256 _era)` do not follow Solidity's naming convention of prefixing private/internal functions with an underscore.

While this does not affect functionality, it deviates from commonly accepted coding standards and may impact code readability and maintainability.

Recommendation

Rename the functions to `_calcWithWeights` and `_updates(uint256 _era)` to align with Solidity's naming conventions for private and internal functions.

Status

Resolved

[Q-03] LiquidStaking - Receive function protection can be bypassed via selfdestruct

Description

In `LiquidStaking.sol`, the `receive()` function implements sender validation to restrict incoming native funds transfers. However, this protection is incomplete as it cannot prevent the contract from forcefully receiving native funds through `selfdestruct()`.

While the current implementation checks for `wastr`, `self-calls`, and `DAPPS_STAKING` as allowed senders, a malicious actor could still force native funds into the contract using selfdestruct, potentially disrupting accounting or internal logic that relies on controlled native funds reception.

Recommendation

Consider adding a comment to document this limitation and ensure that contract logic does not rely solely on native funds balance validation. If strict native funds balance

control is required, implement additional accounting mechanisms separate from the contract's actual balance.

Status

Acknowledged

Centralisation

The Algem Liquid Staking 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 Algem Liquid Staking 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