#Hashlock.

Algem

(LiquidStaking v1.5)

SMART CONTRACT

Security Audit

Platform

ASTR

hashlock.com.au

JAN 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 Algem team partnered with Hashlock to conduct a security audit of their LiquidStaking v1.5 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.

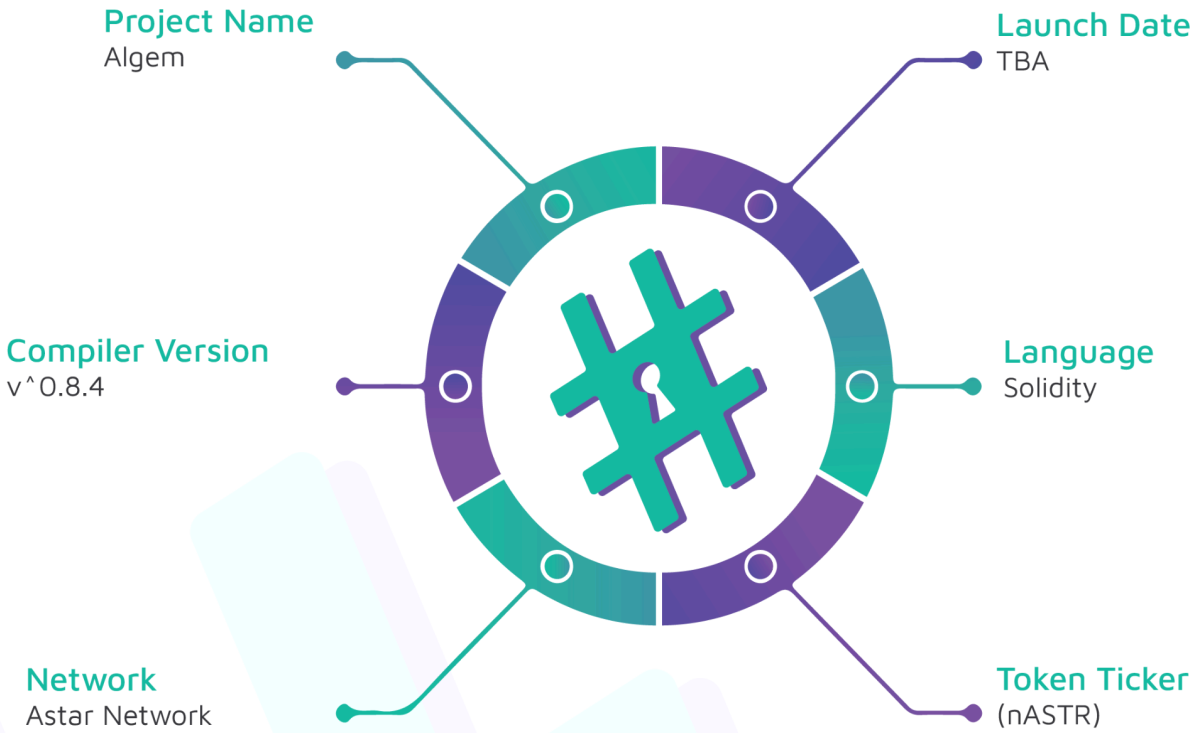
Project Context

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.

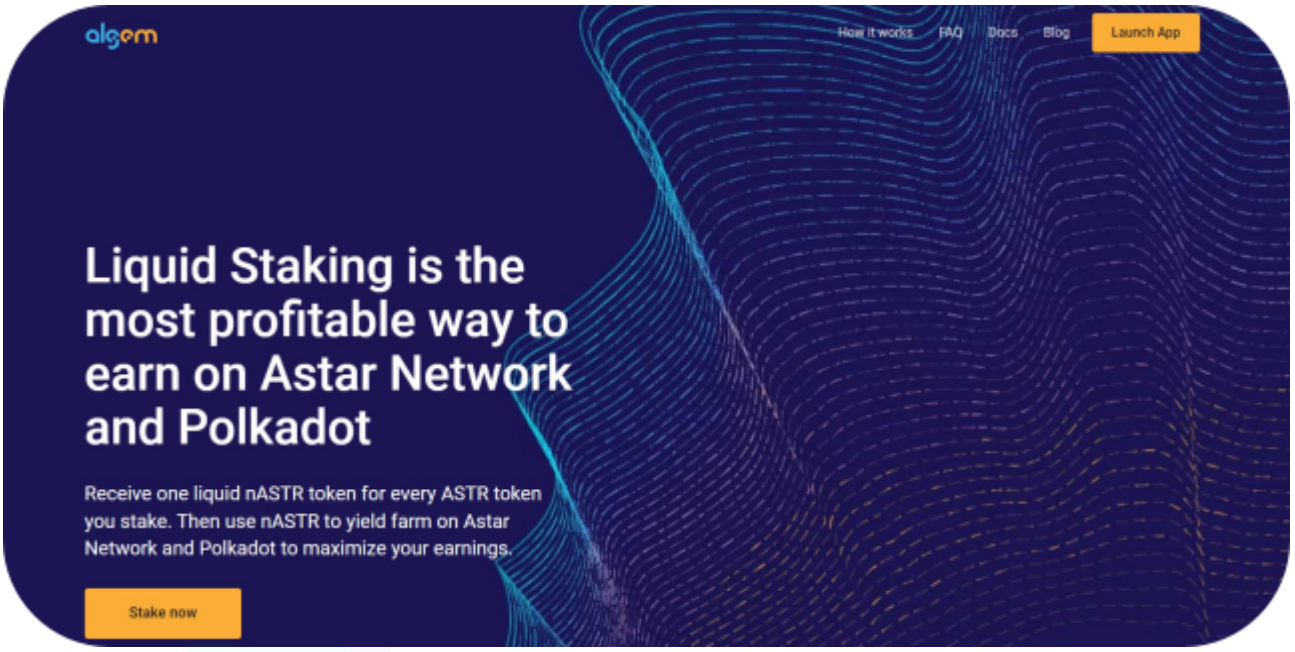
Simply put, you don't have to choose between staking and yield farming with your Astar tokens. You can do both.

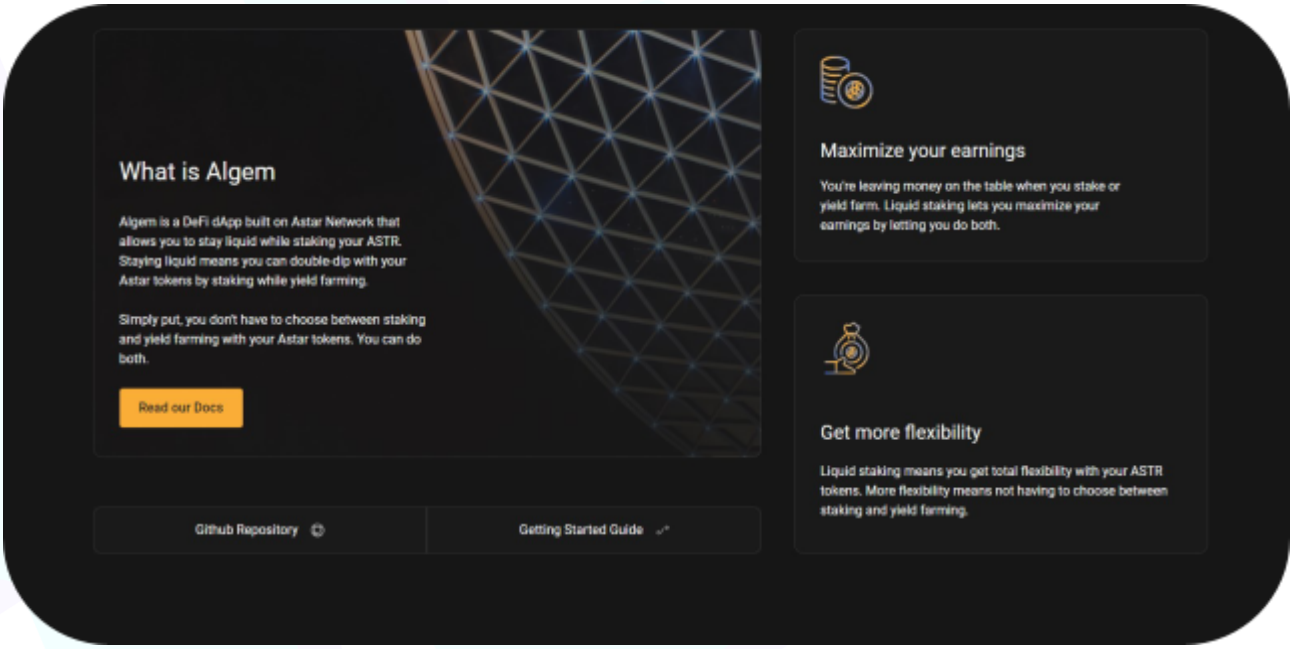
Project Name: Algem
Compiler Version: ^0.8.4
Website: https://www.algem.io/

Visualised Context:



Project Visuals:





Audit scope

We at Hashlock audited the solidity code within the Algem 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	Algem Protocol Smart Contracts
Platform	Solidity
Audit Date	Jan, 2024
Contract 1	LiquidStakingMain.sol
Contract 1 MD5 Hash	b9ff5382d105e12715933a34682d2c06
Contract 2	LiquidStakingAdmin.sol
Contract 2 MD5 Hash	58f7ac8ebb54039e357f18c66fb3540c

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.

All vulnerabilities initially identified have now been resolved and acknowledged.

Hashlock found:

5 High severity vulnerabilities

2 Medium severity vulnerabilities

3 Low severity vulnerabilities

12 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.*

Code Quality

This Audit scope involves the 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.

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 projects 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] Loss of funds due to active dapps without addresses

Description

Te issue lies in the fact that the 'setDappsList' function sets a list of Dapps, but these are not treated as "real used Dapps" in other parts of the system. The 'addDapp' function, which introduces Dapps and sets them to active, does not keep track of the list. Consequently, it is possible to have active Dapps without associated addresses.

Here is an example of the flow that leads to the vulnerability:

1. The manager calls 'setDappsList' with [A, B].
2. The manager then calls 'toggleDappAvailability' with (A).
3. As a result, Dapp A becomes active but does not have an associated address.

The correct flow should involve the 'addDapp' function, where the manager calls 'addDapp(Aname, Aaddress)' first. This ensures that Dapp A is correctly set as active and associated with an address.

Impact

This vulnerability allows users to stake on active Dapps without associated addresses, potentially leading to unexpected behavior in the system.

Recommendations

Remove the `setDappsList` function and keep track of new added dapps to the list within the `addDapp` function.

Status

Resolved

[H-02] Users unable to unstake

Description

For every Dapp (utility) to function correctly, it should be initialized by the LiquidStaking contract with 'haveUtility' set to true, 'isActive' set to true, and 'dappAddress' not equal to zero. The current setup only requires the 'haveUtility' condition during unstaking, and not during staking. This means that users can stake on Dapps with 'haveUtility' set to false, rendering them unable to unstake and resulting in potential loss of funds.

Let's consider the following scenario: the manager adds a new Dapp using the 'addDapp' function, resulting in a Dapp that has not been initialized, thus setting 'haveUtility' to false. In this situation, users can stake on the Dapp but are unable to unstake

Impact

Users are loosing their funds because they have staked into a dapp with 'haveUtility' to false but they can not unstake due to the require:

```
require(haveUtility[_utilities[i]], "Unknown utility");
```

Recommendations

I would suggest adding the same require for staking so users can not stake to unknown utilities and also set the haveUtility value to true when the MANANGER adds a new dapp in LiquidStaking.sol.

Status

Resolved

[H-03] Users unable to stake due not possible to add new dAPPs

Description

The function addDapp which implements an onlyRole(MANAGER) modifier is used to add new dapps to the protocol. This function implements a require:

```
require(dapp.dappAddress != address(0), "Dapp is already added");
```

The problem is that when adding a new dapp dapp.dappAddress will always be address(0) so the tx will always revert.

Impact

No dapps can be added so the protocol does not work.

Recommendations

Change the require condition to == address(0)

Status

Resolved

[H-04] Centralization risk

Description

The contract LiquidStakingAdmin.sol implements a function called withdrawOverage

```
    /// @notice Withdraw rewards overage. Calculates offchain.
    ///         Formed when users use their nASTR tokens in defi protocols bypassing algem-adapters.
    function withdrawOverage(uint256 amount) external onlyRole(MANAGER) {
        rewardPool -= amount;
        payable(msg.sender).sendValue(amount);
    }
```

It can be seen that this function allow the MANAGER to withdraw any amount of ETH from the rewardPool. The NATSPEC says: Calculates offchain. Formed when users use their nASTR tokens in defi protocols bypassing algem-adapters.

It is not necessary that the manager executes a malicious act but that the offchain computation is wrongly done to withdraw more ETH than expected so that the rest of the protocol does not operate as expected neither.

Impact

A malicious act or an offchain mistake can lead to users loosing funds.

Recommendations

Compute onchain the rewards overage by tracking them in the contract so no offchain infrastructure should be trusted.

Status

Resolved

[H-05] Dapp claims can be lost

Description

In the current implementation, when `_claimDapp` is executed with lastUpdatedEra set to a specific value (let's use the example of 1), the function initiates the claiming process starting from that specified era and continuing up to the current era.

For better understanding, consider the following scenario:

1. The current era is 5, and the sync function is called, triggering the execution of _updates(5).
2. Subsequently, _claimDapp(5) is invoked.
3. Within the execution of _claimDapp(5), lastUpdatedEra is was to 1, causing the function to attempt claiming from era 1 to era 5.
4. Assuming era 1 claims successfully, but eras 2 and 3 encounter failures while eras 4 and 5 are claimed successfully.
5. Due to the presence of a try-catch block, the call does not revert, providing a safety net during execution.
6. After the execution, lastUpdated is updated to the current era (5), signifying a successful execution.
7.

However, despite the overall success, eras 2 and 3 failed to claim funds for unknown reasons.
It is important to note that the subsequent sync call in, for example, era 10 triggers _claimDapp, but it begins claiming from era 5. Consequently, any failed claims from era 2 and 3 are irrecoverable, leading to a permanent loss of funds.

Impact

The claims of certain eras that fails can not be claimed during next eras so these claims are lost

Recommendations

Change the mechanism of try-catch and lastUpdated to be able to claim failing claims of past eras.

Status

Resolved

Medium issues

[M-01] Function can overflow and staking/unstaking is not correctly tracked.

Description

The function _updateSubperiodStakes within the LiquidStakingMain contract is called when staking and when unstaking, the difference is that when staking the parameter _amount is positive and when unstaking its negative. The function logic is the following one:

```
/// @dev Updates subperiod stakes. Increase in stake case and decrease if unstake
    /// @param _amount => amount to increase or decrease stake size. Positive if its a stake and negative if its an unstake.
    function _updateSubperiodStakes(int256 _amount) internal {
        uint256 currentPeriodNumber = currentPeriod();
```

```
        Period storage period = periods[currentPeriodNumber];

        if (voteSubperiod()) {
            period.voteStake = uint256(int256(period.voteStake) + _amount);
            period.buidAndEarnStake = uint256(int256(period.buidAndEarnStake) + _amount);
            periodsStakes[msg.sender][currentPeriodNumber][0] =
uint256(int256(periodsStakes[msg.sender][currentPeriodNumber][0]) + _amount);
            periodsStakes[msg.sender][currentPeriodNumber][1] =
uint256(int256(periodsStakes[msg.sender][currentPeriodNumber][1]) + _amount);
        } else {
            period.buidAndEarnStake = uint256(int256(period.buidAndEarnStake) + _amount);
            periodsStakes[msg.sender][currentPeriodNumber][1] =
uint256(int256(periodsStakes[msg.sender][currentPeriodNumber][1]) + _amount);
        } // prettier-ignore

        require(
            period.voteStake <= period.buidAndEarnStake,
            "Unstake size too big"
        );
    }
```

This function can overflow in the following scenario:

1. User stakes during a non voting subperiod so the previous function goes through the else statment.
2. User unstakes during a voting period so the previous function now goes through the if statment.
3. As a result the following statment overflows:
   periodsStakes[msg.sender][currentPeriodNumber][0] =
   uint256(int256(periodsStakes[msg.sender][currentPeriodNumber][0]) + _amount);

This is because this operation is negative int256(periodsStakes[msg.sender][currentPeriodNumber][0]) + _amount) because periodsStakes[msg.sender][currentPeriodNumber][0] is equal to 0 as it has not been modified during the staking action and _amount is now negative representing an unstaking action, the result is a negative number that is casted from int256 to uint256 leading to an overflow.

Impact

The tracking of staking/unstaking does not behaves as expected in an overflow scenario.

Recommendations

Add `periodsStakes[msg.sender][currentPeriodNumber][0] = uint256(int256(periodsStakes[msg.sender][currentPeriodNumber][0]) + _amount);` in the else statment of change the logic to handle the mentioned case.

Status

Resolved

[M-02] Reentrancy

Description

The functions `stake` and `unstake` in the `LiquidStakingMain.sol` contract allow Reentrancy attacks to take place.

```
  if (_immediate) {
      require(
          unstakingPool >= _amount,
          "Unstaking pool drained!"
      );
      uint256 fee = _amount / 100; // 1% immediate unstaking fee
      totalRevenue += fee;
      unstakingPool -= _amount;
      payable(msg.sender).sendValue(_amount - fee);
```

```
// send back the diff
   if (value > stakeAmount) payable(msg.sender).sendValue(value - stakeAmount);
```

Impact

An attacker can perform a reentrancy attack to break the expected behaviour of the protocol. For example, if a reentrancy attack is done within the unstake function the tracking of `totalUnstaked` is altered.

Recommendations

Follow the CEI pattern (check - effect - interactions) and more the sendValue after executing all the changes in the contract state.

Status

Resolved

Low issues

[L-1] Empty Function Body - Consider commenting why

Instances (1):

File: LiquidStakingMain.sol

```
300:     ) external onlyRole(MANAGER) updateRewards(_user, _utilities) {}
```

Status

Resolved

[L-2] Array indices should be referenced via enums rather than via numeric literals

Instances (25):

File: LiquidStakingMain.sol

```
81:         eraBuffer[0] += stakeAmount;

203:             eraBuffer[1] += totalUnstaked;

345:             eras + eraBuffer[0] * (eras - 1) - eraBuffer[1] * (eras - 1);

345:             eras + eraBuffer[0] * (eras - 1) - eraBuffer[1] * (eras - 1);

354:             uint256 nftRevenue = (rewardsK * erasData[1]) /
```

```
356:            uint256 defaultRevenue = rewardsK * REVENUE_FEE * (allErasBalance - erasData[0]) / (100
* REWARDS_PRECISION);

368:        (eraBuffer[0], eraBuffer[1]) = (0, 0);

368:        (eraBuffer[0], eraBuffer[1]) = (0, 0);

562:            nftDistr.updateUser(_utility, _user, lastUpdated - 1, userData[0]);

569:        if (userData[1] == 0) return;

572:        dapps[_utility].stakers[_user].rewards += userData[1];

573:        totalUserRewards[_user] += userData[1];

574:        emit HarvestRewards(_user, _utility, userData[1]);

586:            periodsStakes[msg.sender][currentPeriodNumber][0] =
uint256(int256(periodsStakes[msg.sender][currentPeriodNumber][0]) + _amount);

586:            periodsStakes[msg.sender][currentPeriodNumber][0] =
uint256(int256(periodsStakes[msg.sender][currentPeriodNumber][0]) + _amount);

587:            periodsStakes[msg.sender][currentPeriodNumber][1] =
uint256(int256(periodsStakes[msg.sender][currentPeriodNumber][1]) + _amount);

587:            periodsStakes[msg.sender][currentPeriodNumber][1] =
uint256(int256(periodsStakes[msg.sender][currentPeriodNumber][1]) + _amount);

590:            periodsStakes[msg.sender][currentPeriodNumber][1] =
uint256(int256(periodsStakes[msg.sender][currentPeriodNumber][1]) + _amount);

590:            periodsStakes[msg.sender][currentPeriodNumber][1] =
uint256(int256(periodsStakes[msg.sender][currentPeriodNumber][1]) + _amount);

657:        return userData[1] + dapps[_utility].stakers[_user].rewards;

687:        (userData[0], ) = nftDistr.getUserEraBalance(

704:                if (userData[0] > 0 && isUnique)

708:                userData[1] +=

721:                if (_isZeroBalanceWithNft) userData[0] = 0;

722:            } else userData[0] = _userBalanceWithNft;
```

Status

Resolved

[L-3] Functions not used internally could be marked external

Instances (2):

File: LiquidStakingAdmin.sol

```
70:     function setMinStakeAmount(uint _amount) public onlyRole(MANAGER) {

95:     function getStaker(string memory _utility, address _user, uint256 _era) public view returns
(uint256 eraBalance_, bool isZeroBalance_, uint256 rewards_, uint256 lastClaimedEra_) {
```

Status

Resolved

Gas Optimizations

[GAS-1] Use selfbalance() instead of address(this).balance

Use assembly when getting a contract's balance of ETH.

You can use selfbalance() instead of address(this).balance when getting your contract's balance of ETH to save gas.
Additionally, you can use balance(address) instead of address.balance() when getting an external contract's balance of ETH.

Saves 15 gas when checking internal balance, 6 for external

Instances (8):

File: LiquidStakingMain.sol

```
331:         uint256 balanceBefore = address(this).balance;

339:         uint256 balanceAfter = address(this).balance;

380:             uint256 balanceBefore = address(this).balance;

388:                 emit ClaimDappSuccess(address(this).balance - balanceBefore, _currentEra);
```

```
398:        uint256 balanceBefore = address(this).balance;

401:            unbondedPool += address(this).balance - balanceBefore;

630:            uint256 balanceBefore = address(this).balance;

637:                uint256 gain = address(this).balance - balanceBefore;
```

Status

Resolved

[GAS-2] Use assembly to check for address(0)

Saves 6 gas per instance

Instances (4):

File: LiquidStakingAdmin.sol

```
50:        require(dapp.dappAddress != address(0), "Dapp is already added");

64:        require(_adistr != address(0), "Zero address error");
```

File: LiquidStakingMain.sol

```
319:        require(_user != address(0), "Zero address alarm!");

461:        require(_user != address(0), "Zero address alarm!");
```

Status

Resolved

[GAS-3] Cache array length outside of loop

If not cached, the solidity compiler will always read the length of the array during each iteration. That is, if it is a storage array, this is an extra sload operation (100 additional extra gas for each iteration except for the first) and if it is a memory array, this is an extra mload operation (3 additional gas for each iteration except for the first).

Instances (1):

File: LiquidStakingMain.sol

```
616:         for (uint256 idx; idx < dappsList.length; idx = _uncheckedIncr(idx)) {
```

Status

Resolved

[GAS-4] Use calldata instead of memory for function arguments that do not get mutated

Mark data types as `calldata` instead of `memory` where possible. This makes it so that the data is not automatically loaded into memory. If the data passed into the function does not need to be changed (like updating values in an array), it can be passed in as `calldata`. The one exception to this is if the argument must later be passed into another function that takes an argument that specifies `memory` storage.

Instances (15):

File: LiquidStakingAdmin.sol

```
16:         string memory _utility // there are a single utility for each dapp

30:     function setDappsList(string[] memory _dappsList) external onlyRole(MANAGER) {

48:     function addDapp(string memory _dappName, address _dappAddr) external onlyRole(MANAGER) {

58:     function toggleDappAvailability(string memory _dappName) external onlyRole(MANAGER) {

95:     function getStaker(string memory _utility, address _user, uint256 _era) public view returns
(uint256 eraBalance_, bool isZeroBalance_, uint256 rewards_, uint256 lastClaimedEra_) {
```

File: LiquidStakingMain.sol

```
63:         string[] memory _utilities,

64:         uint256[] memory _amounts

132:         string[] memory _utilities,
```

```
133:        uint256[] memory _amounts,

213:        string[] memory _utilities,

214:        uint256[] memory _amounts

299:        string[] memory _utilities

306:        string memory _utility,

316:        string memory _utility,

653:        string memory _utility,
```

Status

Resolved

[GAS-5] For Operations that will not overflow, you could use unchecked

[GAS-6] Use Custom Errors

Source
Instead of using error strings, to reduce deployment and runtime cost, you should use Custom Errors. This would save both deployment and runtime cost.

Instances (21):

File: LiquidStakingAdmin.sol

```
18:        require(msg.sender == address(distr), "Allowed only for NDistributor");

31:        require(_dappsList.length != 0, "Empty array");

37:        require(bonusRewardsPool > 0, "bonusRewardsPool is emply");

50:        require(dapp.dappAddress != address(0), "Dapp is already added");

64:        require(_adistr != address(0), "Zero address error");

71:        require(_amount > 0, "Should be greater than zero!");

79:        require(totalRevenue >= _amount, "Not enough funds in revenue pool");
```

File: LiquidStakingMain.sol

```
18:         require(_utilities.length > 0, "No one utility selected");

72:             require(isActive[_utilities[i]], "Dapp not active");

73:             require(_amounts[i] >= minStakeAmount, "Not enough stake amount");

78:         require(stakeAmount > 0, "Incorrect amounts");

79:         require(value >= stakeAmount, "Incorrect value");

141:             require(haveUtility[_utilities[i]], "Unknown utility");

260:         require(withdrawal.eraReq != 0, "Withdrawal already claimed");

266:         require(unbondedPool >= val, "Unbonded pool drained!");

290:         require(_era > lastUpdated && _era <= currentEra(), "Wrong era range");

319:         require(_user != address(0), "Zero address alarm!");

461:         require(_user != address(0), "Zero address alarm!");

518:         require(!isPartner[msg.sender], "Claim not allowed for partner pools");

530:                 require(rewardPool >= _amounts[i], "Rewards pool drained");

541:         require(transferAmount > 0, "Nothing to claim");
```

Status

Resolved

[GAS-7] Long revert strings

Instances (1):

File: LiquidStakingMain.sol

```
518:         require(!isPartner[msg.sender], "Claim not allowed for partner pools");
```

Status

Resolved

[GAS-8] Functions guaranteed to revert when called by normal users can be marked payable

If a function modifier such as `onlyOwner` is used, the function will revert if a normal user tries to pay the function. Marking the function as `payable` will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided.

Instances (9):

File: LiquidStakingAdmin.sol

```
30:     function setDappsList(string[] memory _dappsList) external onlyRole(MANAGER) {

36:     function withdrawBonusRewards() external onlyRole(MANAGER) {

48:     function addDapp(string memory _dappName, address _dappAddr) external onlyRole(MANAGER) {

58:     function toggleDappAvailability(string memory _dappName) external onlyRole(MANAGER) {

63:     function setAdaptersDistributor(address _adistr) external onlyRole(MANAGER) {

70:     function setMinStakeAmount(uint _amount) public onlyRole(MANAGER) {

78:     function withdrawRevenue(uint256 _amount) external onlyRole(MANAGER) {

88:     function withdrawOverage(uint256 amount) external onlyRole(MANAGER) {
```

File: LiquidStakingMain.sol

```
289:     function sync(uint _era) external onlyRole(MANAGER) {
```

Status

Resolved

[GAS-9] ++i costs less gas than i++, especially when it's used in for-loops (--i/i-- too)

Saves 5 gas per loop

Instances (9):

File: LiquidStakingMain.sol

```
41:         for (uint256 i; i < l; i++) _harvestRewards(_utilities[i], _user);

46:         for (uint256 i; i < l; i++)

71:         for (uint256 i; i < utilitiesLength; i++) {

99:         for (uint256 i; i < utilitiesLength; i++) {

140:         for (uint256 i; i < utilitiesLength; i++) {

236:         for (uint i; i < l; i++) {

242:         for (uint256 i; i < l + 1; i++) {

249:         for (uint256 i; i < l; i++)

523:         for (uint256 i; i < l; i++) {
```

Status

Resolved

[GAS-10] Use shift Right/Left instead of division/multiplication if possible

Instances (3):

File: LiquidStakingMain.sol

```
180:                 if (lastUnstaked * 10 + (withdrawBlock * 10) / 4 > era * 10) {

181:                     _lag = lastUnstaked * 10 + (withdrawBlock * 10) / 4 - era * 10;
```

```
411:        if (_era * 10 < lastUnstaked * 10 + (withdrawBlock * 10) / 4) return;
```

Status

Resolved

[GAS-11] Splitting require() statements that use && saves gas

Instances (1):

File: LiquidStakingMain.sol

```
290:        require(_era > lastUpdated && _era <= currentEra(), "Wrong era range");
```

Status

Resolved

[GAS-12] Use != 0 instead of > 0 for unsigned integer comparison

Instances (16):

File: LiquidStakingAdmin.sol

```
37:        require(bonusRewardsPool > 0, "bonusRewardsPool is emply");
```

```
71:        require(_amount > 0, "Should be greater than zero!");
```

File: LiquidStakingMain.sol

```
18:        require(_utilities.length > 0, "No one utility selected");
```

```
78:        require(stakeAmount > 0, "Incorrect amounts");
```

```
103:            if (_amounts[i] > 0) {

143:            if (_amounts[i] > 0) {

200:          if (totalUnstaked > 0) {

347:          if (allErasBalance > 0) {

411:        if (_era * 10 < lastUnstaked * 10 + (withdrawBlock * 10) / 4) return;

489:          staker.isZeroBalance[_era] = _amount > 0 ? false : true;

524:            if (_amounts[i] > 0) {

541:        require(transferAmount > 0, "Nothing to claim");

561:            dapps[_utility].stakers[_user].isZeroBalance[lastUpdated] = userEraBalance > 0 ? false :
true; // prettier-ignore

702:            if (userEraBalance > 0) {

704:                if (userData[0] > 0 && isUnique)

726:            if (_userNextEraFee > 0) userEraFee = _userNextEraFee;
```

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 Akgem project seems to have a sound and well tested code base. 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.