hashlock.

Security Audit

Algem Liquid Farming V3 (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

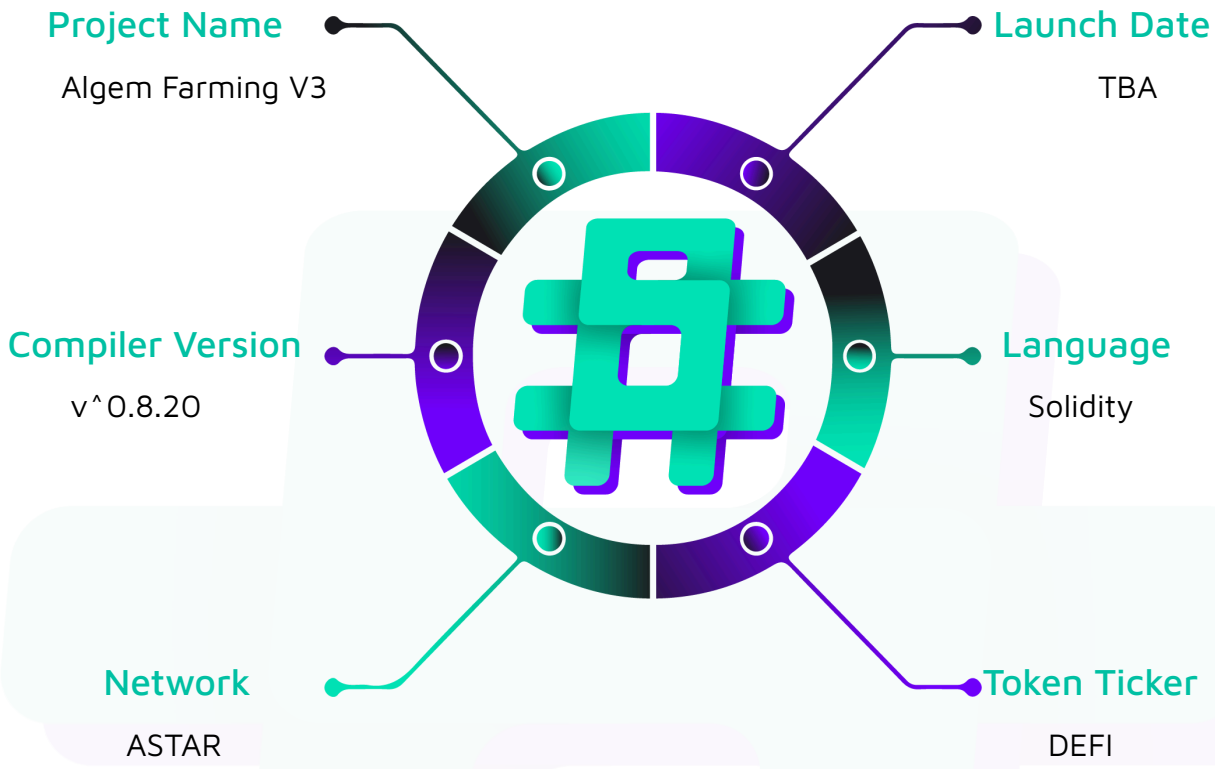
The Algem team partnered with Hashlock to conduct a security audit of their smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

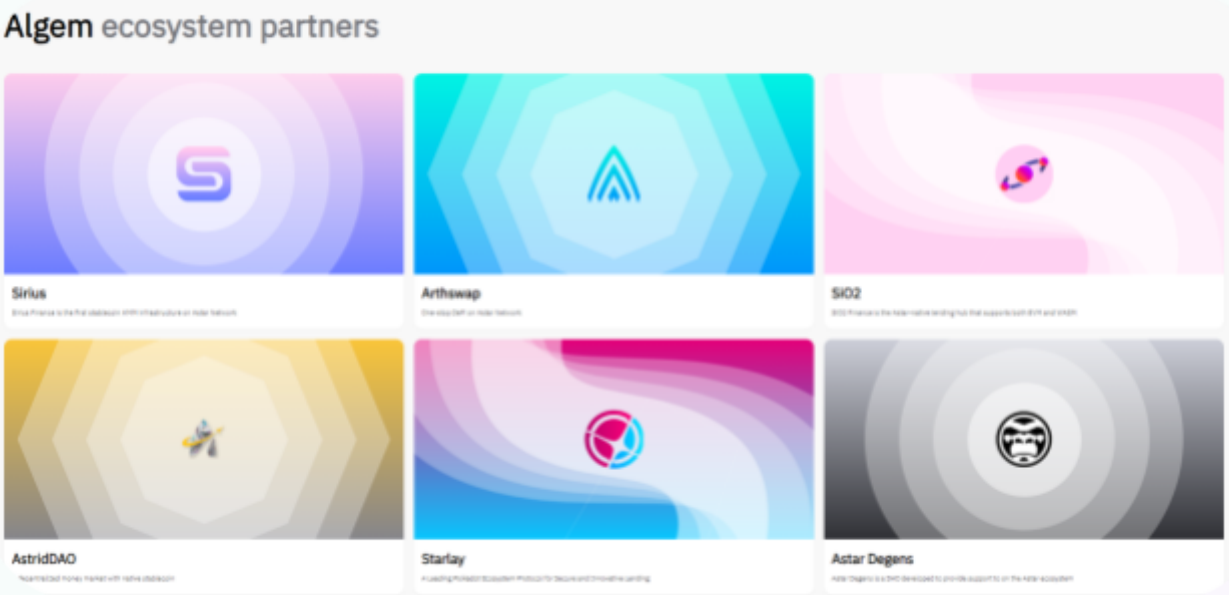
Algem.io is a decentralized liquid staking platform built on the Polkadot ecosystem, allowing users to stake assets while maintaining liquidity through Web3 technologies. It enables participants to earn staking rewards without locking up their tokens, offering flexibility to engage in DeFi activities 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
Project Type: DeFi
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 Farming V3 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 Farming V3 Smart Contracts
Platform	ASTAR / Solidity
Audit Date	December, 2024
Contract 1	LFxKyoV3Vault0.sol
Contract 1 MD5 Hash	6195861e4d5df9c40e0e4ceafa1f01b4
Contract 2	LFxKyoV3Pool.sol
Contract 2 MD5 Hash	c06935ea9628777fdd8d622f2634601b
Contract 3	LFxSonusV3Pool.sol
Contract 3 MD5 Hash	9007b40b04ea22b662e59997c5c6945c
Contract 4	LFxSonusV3Vault0.sol
Contract 4 MD5 Hash	152e66761776710c8b09d7102202c354

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 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:

3 High severity vulnerabilities

5 Medium severity vulnerabilities

3 Low severity vulnerabilities

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
LFxKyoV3Vault0.sol, LFxSonusV3Vault0.sol - Allows users to: - Deposit tokens/WETH - Redeem the deposit - Claim rewards in ALGM - Liquidations	**Contract achieves this functionality.**
LFxKyoV3Pool.sol, LFxSonusV3Pool.sol - Periphery contract used to: - Add vaults - Remove vaults - Set Vault ALGM Shares	**Contract achieves this functionality.**

Code Quality

This audit scope involves the smart contracts of the Algem Farming V3 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 ALGM Farming V3 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

High

[H-01]Pool#addALGM- Infinite Loop Vulnerability Leading to Protocol DoS.

Description

The `addALGM()` function is vulnerable to an infinite loop condition that can cause a Denial of Service (DoS) in the protocol.

The issue stems from a missing increment in a critical loop section, which can permanently damage the protocol's state and calculations when triggered with specific parameters.

Vulnerability Details

```
function addALGM(uint256 _amount) external {

            if (_amount > 0) IERC20(ALGM).safeTransferFrom(msg.sender, address(this),
_amount);

        else return;

        for (uint8 i = 0; i < vaultsCount;) {

            uint256 amount = _amount * shares[i] / 100;

            if (amount == 0) {

                lastDistributed[i]++;

                continue;

            }

            IERC20(ALGM).safeIncreaseAllowance(vaults[i], amount);

                        lastDistributed[i] = ILFVault(vaults[i]).addALGMRewards(amount,
lastDistributed[i]);

            emit DistributedALGM(i, vaults[i], amount);
```

```
            unchecked {
                ++i;
            }
        }
    }
```

Given the following conditions:

- vaultsCount = 1
- sharesVaultsCount = 10
- _amount = 1 wei

The `addALGM()` function makes different checks:

- If the amount is bigger than 0, transfer the amount to the pool contract
- Creates a loop, and the iterations are based on the amount of the `vaultsCount` variable (1 at this moment)
- Creates the amount variable using this calculations: `_amount * shares[i] / 100;`

  Converting this in numbers is: 1 * 10 / 100 = 0
- Check if the amount is equal to 0 and if it is, increment the `lastDistributed`.

At this point, there is no incrementing on the i, which will remain on 0 and restart the iteration from the beginning, incrementing only the `lastDistributed[i]` variable.

The loop continues and the `lastDistributed[i]` arrives at a big number, that will never be reached by the protocol anymore, and leads to big problems in other calculations from other contracts also. At this point, funds from users and protocol integrity are compromised.

Proof of Concept

An example of a test that simulates the bug:

```
function test_addOneWeiALGMGoesInDoS()public{
        v3pool.setVaultShare(0, 10);
```

```
        vm.warp(block.timestamp + 86400);

        address account1 = vm.addr(1);

        algm.mint(account1, 1);



        vm.startPrank(account1);

        algm.approve(address(v3pool), 1 ether);

        v3pool.addALGM(1);

        vm.stopPrank();

    }
```

Impact

Protocol DoS.

Recommendation

1) Add `onlyOwner` modifier

2) increment the i before the continue statement

```
    if (amount == 0) {

        lastDistributed[i]++;

        unchecked { ++i; }

        continue;

    }
```

3) Add a minimum amount check to prevent small amounts

4) Consider implementing a circuit breaker pattern

Status

Resolved

[H-02]LFxKyoV3Vault0#redeem- Token Inflation on Vault Expiration Due to Incorrect Redeem Calculation

Description

The redeem function has a logic error in the calculation of token amounts when the vault expires.

The calculation uses a decreasing `lWRAPPED.totalSupply()` value, causing each subsequent user to receive progressively more KYO tokens than they should.

Vulnerability Details

```
    function redeem() external nonReentrant updater {

        _claim(msg.sender, false);



        Position storage p = positions[msg.sender];

        uint256 tokenToPay;



        if (p.finish != FINISH) {

            //liquidated

            tokenToPay = p.token;

        } else if (expired) {

            tokenToPay = totalTokenBalance * p.wrapped / lWRAPPED.totalSupply();

        } else {

         . . .

        if (p.wrapped > 0) {

            totalWRAPPEDBalance -= p.wrapped;

            lWRAPPED.burn(msg.sender, p.wrapped);

            IWETH9(WRAPPED).withdraw(p.wrapped);

            payable(msg.sender).sendValue(p.wrapped);
```

```
        }

        . . .

    }
```

Let's analyze this scenario with 3 users who each deposited 50 KYO and 50 ETH:

Initial state:

- totalTokenBalance = 150 KYO
- Each user's p.wrapped = 50
- Initial lWRAPPED.totalSupply = 150

Redemption sequence:

1. The first user redeems:
    - 150 * 50 / 150 = 50 KYO
    - `lWRAPPED.totalSupply` decreases to 100
2. The second user redeems:
    - 150 * 50 / 100 = 75 KYO
    - `lWRAPPED.totalSupply` decreases to 50
3. The third user redeems:
    - 150 * 50 / 50 = 150 KYO

Total KYO paid out: 275 KYO from a pool of 150 KYO

The error occurs because `lWRAPPED` tokens are burned after each redeem, decreasing `totalSupply`, and inflating every calculation.

Proof of Concept

An example of a test that simulates the bug:

```
    function test_multipleRedeem()public{

        vm.warp(block.timestamp + 86400);



        address account1 = vm.addr(1);
```

```
        address account2 = vm.addr(2);
        address account3 = vm.addr(3);

        // Account 1
        vm.deal(account1, 1000 ether);
        deal(address(v3vault.pairToken()), account1, 1000 ether);

        // Account 2
        vm.deal(account2, 1000 ether);
        deal(address(v3vault.pairToken()), account2, 1000 ether);

        // Account 3
        vm.deal(account3, 1000 ether);
        deal(address(v3vault.pairToken()), account3, 1000 ether);

        // Account 1 deposit
        vm.startPrank(account1);
        IERC20(address(v3vault.pairToken())).approve(address(v3vault), 1000 ether);
        v3vault.deposit{value:50 ether}(50 ether);
        vm.stopPrank();

        // Account 2 deposit
        vm.startPrank(account2);
        IERC20(address(v3vault.pairToken())).approve(address(v3vault), 1000 ether);
        v3vault.deposit{value:50 ether}(50 ether);
        vm.stopPrank();
```

```
        // Account 3 deposit
        vm.startPrank(account3);
        IERC20(address(v3vault.pairToken())).approve(address(v3vault), 1000 ether);
        v3vault.deposit{value:50 ether}(50 ether);
        vm.stopPrank();



        vm.warp(block.timestamp + v3vault.FINISH());

        vm.startPrank(account1);
        v3vault.redeem();
        vm.stopPrank();



        vm.startPrank(account2);
        v3vault.redeem();
        vm.stopPrank();

        vm.startPrank(account3);
        v3vault.redeem();
        vm.stopPrank();
    }
```

Impact

Protocol pays out more tokens than it has (275 KYO vs 150 KYO available); Later users receive significantly more tokens than early redeemers; Protocol becomes insolvent; Complete loss of funds for users who redeem last.

Recommendation

Change the calculation methods

Status

Resolved

[H-03]LFxKyoV3Vault0#withdraw, redeem- Token Inflation Through Sequential Withdraw-Redeem Exploitation

Description

A user can exploit the protocol to receive an inflated amount of KYO tokens by calling `withdraw()` followed by `redeem()` at vault expiration.
The root of the problem is the same of **[H-02]**.

Vulnerability Details

```
    function withdraw(uint256 _amount) external nonReentrant {

    if (!expired) {

        revert Expiration();

    }


    if (lWRAPPED.balanceOf(msg.sender) < _amount || totalWRAPPEDBalance < _amount) {

        revert InvalidAmount();

    }


    totalWRAPPEDBalance -= _amount;

    lWRAPPED.burn(msg.sender, _amount);


    IWETH9(WRAPPED).withdraw(_amount);

    payable(msg.sender).sendValue(_amount);
```

```
    emit Withdraw(msg.sender, _amount);
}



function redeem() external nonReentrant updater {
    _claim(msg.sender, false);
    Position storage p = positions[msg.sender];
    uint256 tokenToPay;
    if (p.finish != FINISH) {
        tokenToPay = p.token;
    } else if (expired) {
        tokenToPay = totalTokenBalance * p.wrapped / lWRAPPED.totalSupply();
        . . .
    }
}
```

Looking at both functions, the core of the problem can be seen. The withdraw function burns `lWRAPPED` tokens and returns ETH to users, directly reducing the total supply. The redeem function then uses this reduced total supply in its calculations, leading to inflated KYO token returns. Let's consider a scenario where users deposit 50 ETH and 50 KYO each into the protocol.

Initial state:

- `totalTokenBalance` = 100 KYO
- Each user's p.wrapped = 50
- `lWRAPPED.totalSupply` = 100

When a user exploits this vulnerability, the attack begins by:

1. User1 calls `withdraw(50)`:
    - Burns 50 lWRAPPED tokens
    - `lWRAPPED.totalSupply` reduces to 50
    - Receives 50 ETH
2. User1 immediately calls `redeem()`:
    - calculates: 100 * 50 / 50 = 100 KYO
    - Receives 100 KYO (entire pool) instead of their fair share of 50 KYO

Proof of Concept

An example of a test that simulates the bug:

```
function test_withdrawAndRedeem() public {

    vm.warp(block.timestamp + 86400);



    address account1 = vm.addr(1);

    address account2 = vm.addr(2);

    address account3 = vm.addr(3);



    // Setup initial balances for Account 1

    vm.deal(account1, 1000 ether);

    deal(address(v3vault.pairToken()), account1, 1000 ether);



    // Setup initial balances for Account 2

    vm.deal(account2, 1000 ether);

    deal(address(v3vault.pairToken()), account2, 1000 ether);



    // Account 1 deposits 50 ETH and 50 KYO

    vm.startPrank(account1);

    IERC20(address(v3vault.pairToken())).approve(address(v3vault), 1000 ether);
```

```
    v3vault.deposit{value:50 ether}(50 ether);

    vm.stopPrank();

    // Account 2 deposits 50 ETH and 50 KYO

    vm.startPrank(account2);

    IERC20(address(v3vault.pairToken())).approve(address(v3vault), 1000 ether);

    v3vault.deposit{value:50 ether}(50 ether);

    vm.stopPrank();



    // Warp to FINISH time

    vm.warp(block.timestamp + v3vault.FINISH());



    // Account 1 executes the exploit

    vm.startPrank(account1);

    v3vault.withdraw(IERC20(lfcfg.lwrapped).balanceOf(account1));

    v3vault.redeem();

    vm.stopPrank();

}
```

Impact

Protocol becomes insolvent; Other users lose their funds

Recommendation

Change the calculation methods

Status

Resolved

Medium

[M-01] LFMaster, Pool, Vault #addALGM,addALGMRewards - Users that call these functions will lose their ALGM tokens permanently

Description

The `addALGM()` function in LFMaster and Pool contracts and the `addALGMRewards()` functions in Vault contract, are currently accessible to any external user without restrictions.

This design allows any user to transfer ALGM tokens to the contract without any recovery possibility, resulting in a potentially permanent loss of funds.

Vulnerability Details

The `addALGM()`, `addALGMRewards()` functions in LFMaster/Pool/Vault contracts have no restrictions on who can call them:

LFMaster.sol

```
    function addALGM(uint256 _pid, uint256 _amount) external {

        Pool storage p = pools[_pid];

        require(p.deadline > getCurrentRound(), "Invalid deadline!");

        ALGM.safeTransferFrom(msg.sender, address(this), _amount);

        p.totalAlloc += _amount;

        p.roundSupply += _amount / (p.deadline - getCurrentRound());

        emit AddALGM(_pid, p.addr, _amount, p.roundSupply);

    }
```

Pool.sol

```
    function addALGM(uint256 _amount) external {

            if (_amount > 0) IERC20(ALGM).safeTransferFrom(msg.sender, address(this),
_amount);

        else return;

        for (uint8 i = 0; i < vaultsCount;) {

            uint256 amount = _amount * shares[i] / 100;

            if (amount == 0) {

                lastDistributed[i]++;

                continue;

            }

            IERC20(ALGM).safeIncreaseAllowance(vaults[i], amount);

                        lastDistributed[i] = ILFVault(vaults[i]).addALGMRewards(amount,
lastDistributed[i]);

            emit DistributedALGM(i, vaults[i], amount);

            unchecked {

                ++i;

            }

        }

    }
```

Vault.sol

```
 function addALGMRewards(uint256 _amount) external {

    ALGM.safeTransferFrom(msg.sender, address(this), _amount);

    //console.log("algmRewardPool", algmRewardPool);

    algmRewardPool += _amount;

    //console.log("algmRewardPool after", algmRewardPool);
```

```
    uint256 tr = totalRounds();
    //console.log("totalRounds", totalRounds());
    for (uint256 i = 1; i <= tr;) {
        algmRewards[i] += _amount / tr;
        emit HarvestALGM(i, _amount / tr);

        unchecked {
            ++i;
        }
    }
}
```

Scenario:

- 1. An uninformed user calls `addALGM()` with `_amount` = 1000 ALGM
- 2. Tokens are transferred to the contract
- 3. The user has no way to recover the tokens

Different from the Admin side, where there is the removePool() function that provides the possibility to retrieve all ALGM Tokens from the contract.

Impact

Permanent loss of ALGM tokens for users

Recommendation

Add `onlyOwner()` modifier.

Status

Resolved

[M-02] LFxKyoV3Vault0#deposit - Single Deposit `Limitation` Due to Round Claiming Logic Error

Description

A logic error in the round-claiming mechanism prevents users from making multiple deposits. The issue stems from an inconsistency in how claiming rounds are updated between farming and ALGM rewards, leading to a state where subsequent deposits are permanently blocked after the first claim operation.

Vulnerability Details

```
function deposit(uint256 _amount) external payable whenNotPaused updater {

        . . .

        Position storage p = positions[msg.sender];

        if(p.roundStart > 0)

                require(p.roundClaimedA == cr - 1 && p.roundClaimedF == cr - 1, "Claim
first");

        . . .

        if (p.start == 0) {

            p.start = block.timestamp;

            p.finish = FINISH;

        }

        if (p.roundStart == 0) {

            p.roundStart = cr;

            if (cr > 1) {

                p.roundClaimedA = cr;

                p.roundClaimedF = cr;

            }

        }
```

```
    . . .

}
```

The vulnerability manifests through the following sequence:

Initial Deposit Mechanism:

- When a user makes their first deposit, roundStart is set to the current round (cr).
- If cr > 1, both `roundClaimedA` and `roundClaimedF` are set to cr.
- For deposits in round 1, these claiming rounds remain uninitialized.

Claiming Process

```
function _claim(address _user, bool _restake) internal updater {

    (uint256[2] memory f, uint256 a) = previewRewards(_user);

    if (f[0] + f[1] > 0) {

        if (f[0] > 0) rewardPool[0] -= f[0];

        if (f[1] > 0) rewardPool[1] -= f[1];

        positions[_user].roundClaimedF = getCurrentRound() - 1;

        IWETH9(WRAPPED).withdraw(f[0]);

        payable(_user).sendValue(f[0]);

        IERC20(pairToken).safeTransfer(_user, f[1]);

        if (f[0] > 0) {

            emit ClaimFarming(_user, WRAPPED, f[0]);

        }

        if (f[1] > 0) {

            emit ClaimFarming(_user, pairToken, f[1]);

        }

    }

    if (a > 0) {

        algmRewardPool -= a;
```

```
            positions[_user].roundClaimedA = getCurrentRound() - 1;

            if (_restake) {

                _restakeALGM(_user, a);

            } else {

                ALGM.safeTransfer(_user, a);

            }

            emit ClaimALGM(_user, a);

        }

    }
```

- The `_claim()` function updates `roundClaimedA` and `roundClaimedF` separately
- Crucially, rounds are only updated when rewards exist
- If a user has farming rewards = 0, `roundClaimedF` remains unchanged
- If a user has ALGM rewards > 0, `roundClaimedA` is updated to `getCurrentRound()` - 1

In this case, the result from internal calls to `previewRewards()` that call internally `previewRoundFarming()` is an array of 2 elements that are both 0.

```
     function previewRoundFarming(address _user, uint256 _round) internal view returns
(uint256[2] memory comms_) {

        uint256 farmingRewardsShare;


        if (_round <= positions[_user].roundClaimedF) return comms_;


          uint256 a = 10_000 * (_round - positions[_user].roundStart) / (totalRounds() -
positions[_user].roundStart);


        if (roundBalance[_round] > 0) {
```

```
        farmingRewardsShare = REWARDS_PRECISION * (10_000 - a) * positions[_user].wrapped
/ roundBalance[_round];

        }

        if (roundALGMStaked[_round] > 0) {

                farmingRewardsShare += REWARDS_PRECISION * a * userALGMBalance[_user] /
roundALGMStaked[_round];

        }

        comms_[0] = farmingRewardsShare * farmingRewards[_round][0] / REWARDS_PRECISION /
10_000;

        comms_[1] = farmingRewardsShare * farmingRewards[_round][1] / REWARDS_PRECISION /
10_000;

    }
```

Looking at this function, the problem is that the `farmingRewards[_round][0]` and `farmingRewards[_round][1]` are both 0 so once multiplied the result is always 0.
The `_claim` function won't be able to update `roundClaimedF` correctly, because the rewards are 0.

Subsequent Deposit Blocking

- The deposit function includes the check:

```
require(p.roundClaimedA == cr - 1 && p.roundClaimedF == cr - 1, "Claim first");
```

Due to the inconsistent update mechanism, after claiming:

- `roundClaimedA` is updated properly
- `roundClaimedF` remains at its previous value

This creates a permanent state where the requirement can never be satisfied

Proof Of Concept

```
    function test_doubleDeposit()public{

        vm.warp(block.timestamp + 86400);
```

```
        address account1 = vm.addr(1);
        address account2 = vm.addr(2);
        // Account 1
        vm.deal(account1, 1000 ether);
        deal(address(v3vault.pairToken()), account1, 1000 ether);

        // Account 2
        vm.deal(account2, 1000 ether);
        deal(address(v3vault.pairToken()), account2, 1000 ether);

        // Account 1 deposit
        vm.startPrank(account1);
        IERC20(address(v3vault.pairToken())).approve(address(v3vault), 1000 ether);
        v3vault.deposit{value:50 ether}(50 ether);
        vm.stopPrank();

        // 5 Days
        vm.warp(block.timestamp + (86400 * 5));

        // Account 2 deposit, claim, deposit
        vm.startPrank(account2);
        IERC20(address(v3vault.pairToken())).approve(address(v3vault), 1000 ether);

        v3vault.deposit{value:50 ether}(50 ether);
        // 2 Days
        vm.warp(block.timestamp + (86400*2));
```

```
        v3vault.claim(false);

        v3vault.deposit{value:50 ether}(50 ether);

        vm.stopPrank();

    }
```

Impact

Users are restricted to making only a single deposit.

Recommendation

Update both `roundClaimedA` and `roundClaimedF` in the `_claim()` function regardless of whether rewards exist

Status

Resolved

[M-03] Vault#addALGMRewards - Calculation and Iteration Errors Prevent Proper ALGM Token Reward Distribution

Description

There are several problems that result in incorrect reward calculations and improper token distribution when the function is called.

Vulnerability Details

```
function addALGMRewards(uint256 _amount, uint256 _round) external returns  (uint256) {

        if (_round > totalRounds()) return _round;

        ALGM.safeTransferFrom(msg.sender, address(this), _amount);

        algmRewardPool += _amount;

        uint256 cr = getCurrentRound();

        uint256 rounds = cr - _round;



        if (rounds == 0) return _round;
```

```
        ++_round;


        uint256 amount = _amount / rounds;


        for (uint256 i = _round; i < cr; i++) {
            algmRewards[_round] += amount;
            unchecked {
                ++_round;
            }
        }
        emit HarvestALGM(_round, _amount);
        return cr;
    }
```

The `addALGMRewards()` function accepts an `_amount` parameter that is added to the `algmRewardPool` variable.

The `algmRewardPool` tracks the total ALGM tokens pending distribution.

Following the function flow, there is a round incrementing and after that, there is a calculation:

```
uint256 amount = _amount / rounds;
```

This calculation intends to split the amount of ALGM added, calling this function, for each round, and this is supported by the fact that there is a for loop that tries to set the amount for each round.

Now the problems are multiple:

1. The initial round increment is redundant and problematic, as it causes the function to skip the first round of reward distribution.

This creates an unintended gap in the reward distribution sequence.

2. `uint256 amount = _amount / rounds;`

This calculation, to obtain the amount to be split for each round is wrong because, at this moment, only the amount passed is split for each round and not the entire algmRewardPool.

3. The for loop, iterating only on the round passed, leaving the rest round without additional rewards.

This is caused by a wrong iteration:

```
for (uint256 i = _round; i < cr; i++) {

      algmRewards[_round] += amount;

}
```

The only iteration is on the current `_round` instead of " i "

Proof Of Concept

```
function test_addALGM()public{



    vm.warp(block.timestamp + 86400);

    v3pool.setVaultShare(0, 10);

    algm.mint(address(this), 1000 ether);

    algm.approve(address(v3pool), 1000 ether);

    v3pool.addALGM(133 ether);

    vm.warp(v3vault.FINISH());

}
```

Impact

Rewards not correctly distributed and round skipping

Recommendation

Modify the `addALGMRewards()` function; Example of a correct implementation could be:

```
    function addALGMRewards(uint256 _amount, uint256 _round) external returns (uint256) {

        if (_round > totalRounds()) return _round;

        ALGM.safeTransferFrom(msg.sender, address(this), _amount);

        algmRewardPool += _amount;

        uint256 cr = getCurrentRound();

        uint256 rounds = cr - _round;



        if (rounds == 0) return _round;

        uint256 amount = algmRewardPool / rounds;

        for (uint256 i = _round; i < cr; i++) {

            algmRewards[i] += amount;

            unchecked {

                ++_round;

            }

        }

        emit HarvestALGM(_round, _amount);

        return cr;

    }
```

Status

Resolved

[M-04] LFxKyoV3Vault0#redeem - missing updater modifier, will not set the right expiration time.

Description

The expiration variable is set using the updater modifier, which is not set on the function. At this point, the function call will always fail due to a required statement, that checks the expiration time.

Vulnerability Details

```
function withdraw(uint256 _amount) external nonReentrant {

    if (!expired) {

        revert Expiration();

    }



    if (lWRAPPED.balanceOf(msg.sender) < _amount || totalWRAPPEDBalance < _amount) {

        revert InvalidAmount();

    }



    totalWRAPPEDBalance -= _amount;

    lWRAPPED.burn(msg.sender, _amount);



    IWETH9(WRAPPED).withdraw(_amount);

    payable(msg.sender).sendValue(_amount);



    emit Withdraw(msg.sender, _amount);

}
```

Impact

Expiration time will never be updated, and the function will always revert.

Recommendation

Add updater modifier.

Status

Resolved

[M-05] LFxKyoV3Vault0#withdraw, redeem - Potential Fund Loss in LFxKyoV3Vault0 Due to Withdraw/Redeem Function Sequencing.

Description

The vulnerability exists in the interaction between the withdraw and redeem functions of the `LFxKyoV3Vault0` contract.

When a user executes these functions in a specific order (withdraw followed by redeem), they can permanently lose access to their pair tokens while only receiving their base tokens.

Vulnerability Details

```
function withdraw(uint256 _amount) external nonReentrant updater {

        if (!expired) {

            revert Expiration();

        }

        if (lWRAPPED.balanceOf(msg.sender) < _amount || totalWRAPPEDBalance < _amount) {

            revert InvalidAmount();

        }

        totalWRAPPEDBalance -= _amount;

        lWRAPPED.burn(msg.sender, _amount);

        IWETH9(WRAPPED).withdraw(_amount);
```

```
        payable(msg.sender).sendValue(_amount);

        emit Withdraw(msg.sender, _amount);

    }
```

```
      function redeem() external nonReentrant updater{

        _claim(msg.sender, false);

        Position storage p = positions[msg.sender];

        uint256 tokenToPay;


       . . .

        if (p.wrapped > 0) {

            totalWRAPPEDBalance -= p.wrapped;

            lWRAPPED.burn(msg.sender, p.wrapped);

            IWETH9(WRAPPED).withdraw(p.wrapped);

            payable(msg.sender).sendValue(p.wrapped);

        }

       . . .

    }
```

The Redeem and the withdraw functions for certain things, work in the same way.

The redeem function first claims the rewards if there are some, and then redeems the tokens from the pool using the lWrapped balance of the user that calls the function and burns it.

At the same time, the withdraw function is used to withdraw only the base token, burning `lwrapped`, and after expiration.

Now at this point, if a user withdraws first and redeems then, will get only the base tokens part, leaving the pair tokens in the vault contract and leading to a loss of funds for the user.

This happens because the withdraw function doesn't handle the pair token transfer to the user.

Proof Of Concept

```
    function test_depositWithdrawRedeem()public{

        v3pool.setVaultShare(0, 10);

        vm.warp(block.timestamp + 86400);

        address account1 = vm.addr(1);

        vm.deal(account1, 1000 ether);

        deal(address(pair), account1, 1000 ether);


        //DEPOSITs VAULT

        vm.startPrank(account1);

        pair.approve(address(v3vault), 100 ether);

        v3vault.deposit{value:7 ether}(44 ether);

        vm.warp(block.timestamp + 1);

        v3vault.update();

        vm.stopPrank();


        vm.warp(v3vault.FINISH()-1);

        generateRewards(65);

        v3vault.update();

        vm.warp(v3vault.FINISH());
```

```
        // Claims From Vault
        vm.startPrank(account1);


        v3vault.withdraw(lwrapped.balanceOf(account1));
        vm.warp(block.timestamp + 1);
        v3vault.update();
        vm.stopPrank();
                            console.log("lwrapped.balanceOf(account1)   after   withdraw",
lwrapped.balanceOf(account1));
        console.log("kyo in vault", pair.balanceOf(address(v3vault)));
        v3vault.redeem();
        vm.warp(block.timestamp + 1);
        v3vault.update();
                              console.log("lwrapped.balanceOf(account1)   after   redeem",
lwrapped.balanceOf(account1));
    }
```

Impact

Users can lose their pair tokens deposited in the vault.

Recommendation

Remove withdraw and keep redeem, could be a good solution.

Otherwise, find a way to handle pair tokens too, in the withdraw function.

Status

Acknowledged

Low

[L-01] LFMaster#lsBonus - Variable never initialized

Description

There are no interactions with the `lsBonus` variable at this moment for any contract.

Status

Resolved

[L-02] LFMaster#lockNFT,unlockNFT - functions don't follow the CEI Pattern.

Description

These functions do not follow the CEI pattern to prevent a reentrancy attack.

Recommendation

Always Follow the CEI pattern to prevent reentrancy attacks

Status

Resolved

[L-03] LFMaster#lockNFT - NFT Allowance Handling.

Description

The `lockNFT()` function in the `LFMaster` contract contains a logical error that results in limited usability.

Vulnerability Details

The `lockNFT()` function contains the following implementation:

```
    function lockNFT(address _nft, uint256 _id) external {

        require(userSlots[msg.sender].nft == address(0), "Already locked!");

        require(IERC721(_nft).ownerOf(_id) == msg.sender, "Not the NFT owner");
```

```
        userSlots[msg.sender] = NFTSlot({nft: _nft, id: _id});

        IERC721(_nft).safeTransferFrom(msg.sender, address(this), _id);

        emit NFTLocked(msg.sender, _nft, _id);

    }
```

The function implements an ownership verification through `require(IERC721(_nft).ownerOf(_id) == msg.sender, "Not the NFT owner");`.

This check creates a limitation in the following scenario:

The function does not support NFT allowances:

- User A approves User B to handle their NFT through the standard ERC721 allowance mechanism
- When User B attempts to call `lockNFT()` with the approved NFT:
    - The function strictly checks for direct ownership using `ownerOf()`
    - The transaction reverts with "Not the NFT owner"

This implementation ignores the standard ERC721 approval mechanism, preventing legitimate use cases where users have been granted permission to manage NFTs on behalf of others.

This second check is also redundant because the ERC721 mechanism already grants this.

Impact

These issues result in limiting the contract functionality.

Recommendation

Remove the strict `ownerOf()` check.

Status

Resolved

Centralisation

The Algem 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 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