#Hashlock.

Algem Crowdloan (DEFI)

SMART CONTRACT

Security Audit

Performed on Contracts:

LiquidCrowdloan.sol

MD5 Hash: 4e9ed6e8cc88f0d42ef13ef11d5e4429

ALGMVesting.sol

MD5 Hash: 978e759341706866e6d8754ea72d785a

Platform

ASTAR

hashlock.com.au

MARCH 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 SOLVED. 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

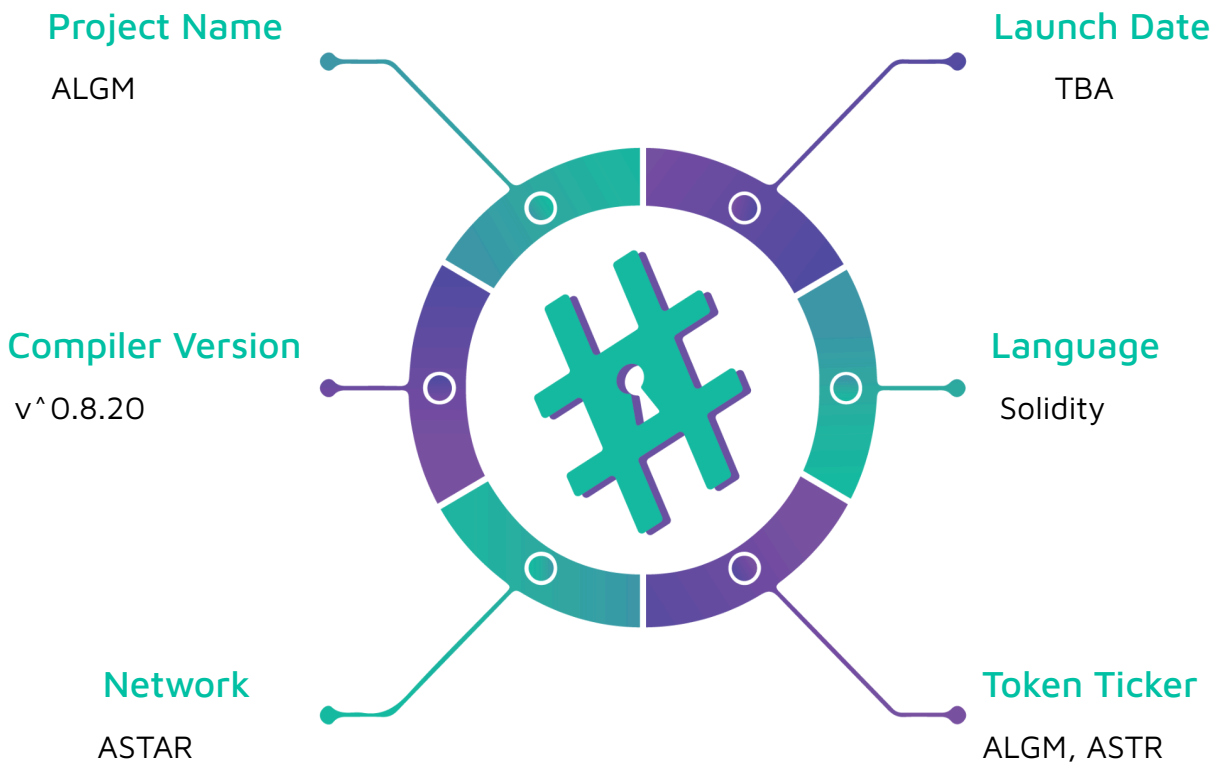
The ALGM team partnered with Hashlock to conduct a security audit of their LiquidCrowdloan.sol, ALGMVesting.sol smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

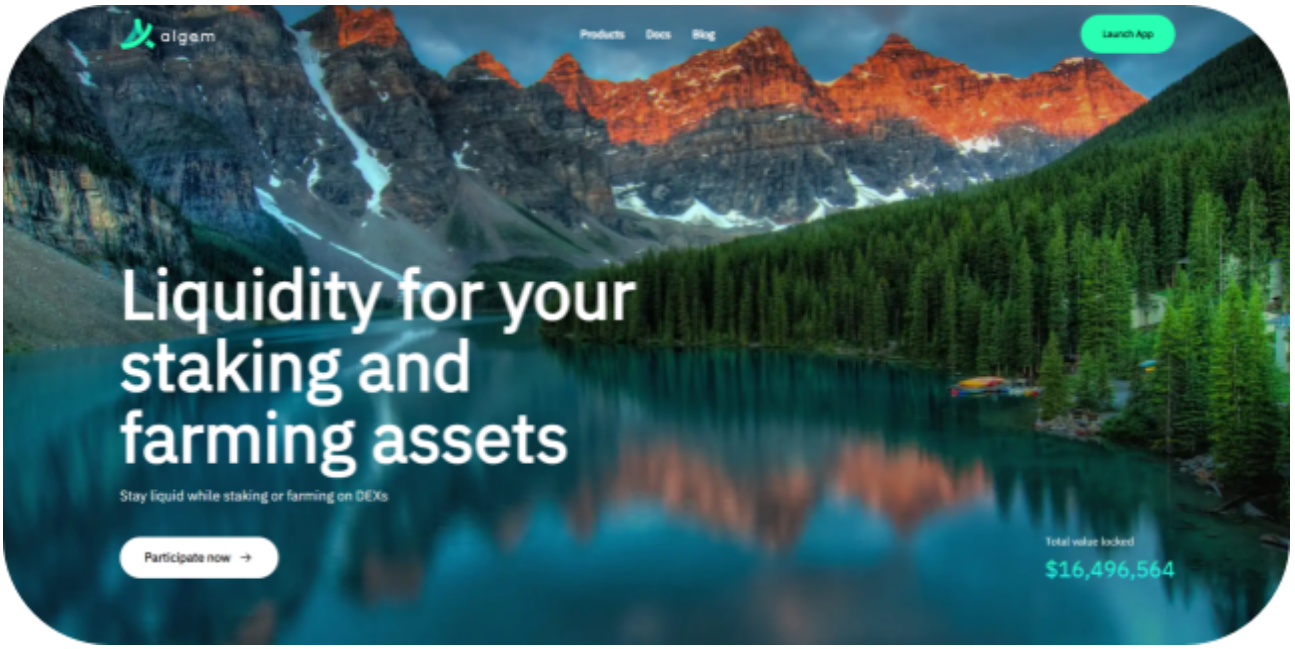
ALGM is a DeFi dApp built on ASTAR Network that allows you to stake/unstake ASTR tokens, get rewards from the stake, and create dedicated vesting.

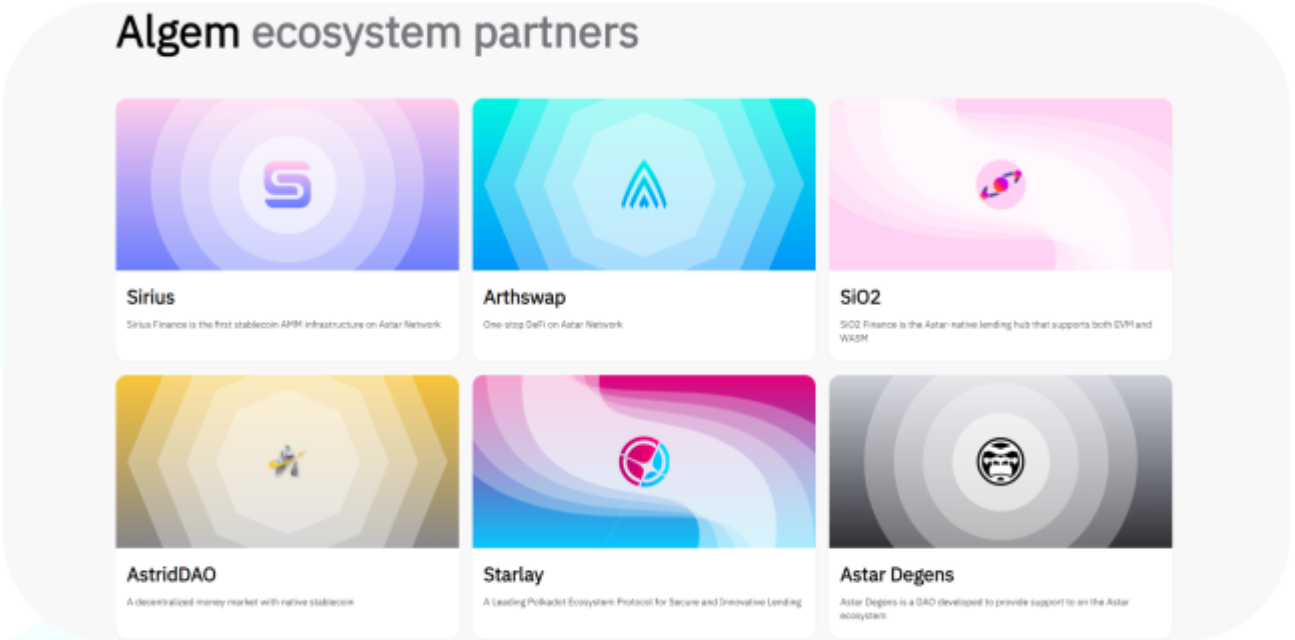
Project Name: ALGM LiquidCrowdloan, ALGMVesting
Compiler Version: ^0.8.20
Website: www.algem.io
Logo:

Visualised Context:



Project Visuals:





Audit scope

We at Hashlock audited the solidity code within the ALGM 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	**ALGM Protocol Smart Contracts**
Platform	**ASTAR / Solidity**
Audit Date	**March, 2024**
Contract 1	LiquidCrowdloan.sol
Contract 1 MD5 Hash	4e9ed6e8cc88f0d42ef13ef11d5e4429
Contract 2	ALGMVesting.sol
Contract 2 MD5 Hash	978e759341706866e6d8754ea72d785a

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 need to be addressed before launch.

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 we have identified have been resolved or acknowledged.

Hashlock found:

3 High severity vulnerabilities

1 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 Behaviours

Claimed Behaviour	Actual Behaviour

LiquidCrowdloan.sol
- Allows users to:
  - Stake ASTR Tokens
  - Create a Vesting
  - Claim Rewards from Vesting
  - Interact with ALGM Token
  - Withdraw the stake when Vesting is end
- Allows admins to:
  - Claim different type of Rewards from the DappsStaking.sol
  - Change the ownership of the contract
	Contract achieves this functionality.

ALGMVesting.sol - Create different type of Vesting - Revoke different type of Vesting. - Interact with ALGM tokens - Add/Delete manager to interact with ALGMVesting contract	**Contract achieves this functionality.**

Code Quality

This Audit scope involves the smart contracts of the ALGM project, as outlined in the Audit Scope section. All contracts, libraries and interfaces mostly follow standard best practices and to help avoid unnecessary complexity that increases the likelihood of exploitation, however some refactoring is required.

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

Audit Resources

We were given the ALGM 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 help us understand 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 Resolved 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] LiquidCrowdloan#claimRewards - Different users can't call claimRewards() function at the same time because there is a risk of insolvency and some users will not receive rewards.

Description

When different users call the `claimRewards()` function at the same time or with not enough time passed between the last call to it, we have a problem of insolvency from the protocol caused by calculations errors.

This is caused by two different bugs: wrong rewards calculations and logic error.

Vulnerability Details

Suppose this scenario:

Two users (User1 and User2) stake 10000 ASTR each.

After 30 days they want to claim rewards.

User1 calls the claimRewards() function, which will interact with the ALGMVesting.

```
function claimRewards() public nonReentrant whenNotPaused {

        uint256 rewards = getUserAvailableRewards(msg.sender);

        . . .

        try vesting.claimVesting(id) {

            emit GlobalClaimSuccess(id);

        } catch Error(string memory reason) {
```

```
            emit GlobalClaimFail(id, reason);

        }

        . . .

        if (!algm.transfer(msg.sender, rewards)) revert TransferError();

        . . .

    }
```

claimRewards() check the amount of available rewards for the msg.sender, by getUserAvailableRewards() function.

In this case(after calc that you can check in ProofOfConcept) rewards for each user are around `208333*10**18 ALGM after 30 days of stake.`

At this point if calculateClaimableQty() function from ALGMVesting is called, we get the amount claimable from the vesting after 30 days:

```
312499*10**18 after 30 days of stake.
```

Now, there is a visible calculation error in one of the two functions that calculate the amount of rewards, because the calculateClaimableQty() function, at this moment, can't satisfy the claim from the two users that stake the same amount of tokens, for the same time.

Logic Error refers to the workflow of the claimRewards() function, that is called twice at the "same time", or with not enough time passed from the last call to it, from different users.

When User1 and User2 call claimRewards() function only one of them calls, will reach the `vesting.claimVesting(id)` to receive rewards in LiquidCrowdloan.

```
function claimRewards() public nonReentrant whenNotPaused {

            . . .

        try vesting.claimVesting(id) {

            emit GlobalClaimSuccess(id);

        } catch Error(string memory reason) {

            emit GlobalClaimFail(id, reason);

        }

        . . .

        if (!algm.transfer(msg.sender, rewards)) revert TransferError();

        . . .

    }
```

When this function is called from User1, will call the `vesting.claimVesting(id)` function:

```
    function claimVesting(uint256 id) external existingVesting(id) {

        . . .

        uint256 claimableQty = _calculateClaimableQty(id);

        require(claimableQty > 0, "Nothing To Claim");

        v.claimedQty += claimableQty;

        . . .

        algmToken.transfer(v.beneficiary, claimableQty);

         . . .

    }
```

This function sends a proportional amount of ALGM tokens to the LiquidCrowdloan contract, based on the time passed.

At this point if User1 and User2 call at the same time the claimRewards() function, only the User1 tx will be able to call and execute the `vesting.claimVesting(id)` function and send ALGM Tokens from the ALGMVesting to LiquidCrowdloan, instead the second tx from User2 will be caught from the try/catch statement in claimRewards() and will not mint anything, because not enough time has passed from the last call.

Proof of Concept

An example of a test that simulates these two errors is shown below.

```
function test_stakeFrom2DifferentAccountsAndGetRewardsDuringVestingPeriod() public {

        vm.prank(user);

        cl.stake{value: 1000 ether}();

        vm.prank(user2);

        cl.stake{value: 1000 ether}();

        vm.prank(owner);

        cl.closeCrowdloan();

        // 30 Days

        vm.warp(30*86400);


        // Check rewards for User1 and User 2 after 30 days

        uint rewardUser1After30Days = cl.getUserAvailableRewards(user);

        uint rewardUser2After30Days = cl.getUserAvailableRewards(user2);

        console.log("Available Rewards User1 after 30 days", rewardUser1After30Days)

        console.log("Available Rewards User2 after 30 days", rewardUser2After30Days);

        console.log("");


        uint amountClaimableAtThisPoint = vesting.calculateClaimableQty(1);
```

```
        console.log("Claimable Amount after 30 days", amountClaimableAtThisPoint);


        // Claim Rewards User
        vm.prank(user);
        cl.claimRewards();


        // Claim Rewards User2
        vm.prank(user2);
        cl.claimRewards();
    }
```

Impact

The wrong amount of rewards sent, will cause Reward Insolvency.

Recommendation

Change the logic behind the rewards calculation.

Status

Resolved

[H-02] LiquidCrowdloan,ALGMVesting#claimRewards,revokeVesting - Manager can revoke vesting at any point in time, and don't send rewards to users.

Description

Users stake on LiquidCrowdloan to get rewards in ALGM tokens, but at any point in time, the owner of the contract can revoke the vesting.

As for [H-01], the error comes from wrong calculations for the rewards.

Vulnerability Details

Let's suppose there are 2 users: User1 stakes 1000 ASTR, User2 stake 1500 ASTR.

The owner of liquidCrowdloan closes the crowdLoan and a new Vesting is created.

After 3 months, the owner decides to delete the vesting created via liquidCrowdloan calling the `revokeVesting` function from ALGMVesting

```
    function revokeVesting(
        uint256 id
    ) external onlyRole(MANAGER) existingVesting(id) {
        . . .
        uint256 claimableQty = _calculateClaimableQty(id);
        v.isRevoked = true;
        v.claimedQty += claimableQty;
        totalAlgmInVestings -= claimableQty;
        algmToken.burn(address(this), v.totalQty - v.claimedQty);

        if (claimableQty > 0) algmToken.transfer(v.beneficiary, claimableQty);
```

```
        emit VestingRevoked(msg.sender, id);
    }
```

After performing some checks, this function checks the claimable quantity at that point in time, and burns the amount that has not matured yet from vesting.

After this, send the rest of the amount to the Vesting beneficiary.

```
Ex:
  Let's suppose we have a vesting of 1000 ALGM for 6 months duration:
  claimableQty_after3Months = 500 ALGM
  tokenBurned = 500 ALGM
  totalAmountSentToLiquidCrowdloan = 500
```

Now vesting is deleted and LiquidCrowdloan has only a portion of the ALGM tokens.

At this point if User1 call the claimRewards() function from LiquidCrowdloan():

```
    function claimRewards() public nonReentrant whenNotPaused {
        uint256 rewards = getUserAvailableRewards(msg.sender);
        if (rewards == 0) revert NoRewards();
        . . .
        try vesting.claimVesting(id) {
            emit GlobalClaimSuccess(id);
        } catch Error(string memory reason) {
            emit GlobalClaimFail(id, reason);
```

```
        }

        . . .

        if (!algm.transfer(msg.sender, rewards)) revert TransferError();

        . . .

    }
```

First of all, check the amount of rewards and, as before [H-01], there is a problem on calculating the amount of rewards.

Then, there is a try/catch statement.

Now the id passed to call vesting.claimVesting(id), "doesn't exist" anymore and we get the error "Vesting Revoked".

At this point, there is no revert in try/catch and the function goes on.

In the end, there is a transfer of ALGM tokens(wrong amount from calc) from LiquidCrowdloan to User1 that will work properly, because the LiquidCrowdloan contract has a portion of the ALGM tokens obtained from call to vesting.revokeVesting(id).

Now User2 wants to do the same and tries to call the claimRewards() function.

This time, User2 doesn't receive anything because now, liquidCrowdloan contract doesn't have as many tokens as needed to send to User2 as rewards, and we get the error "ERC20: transfer amount exceeds balance" .

Proof of Concept

An example of a test that simulates these errors is shown below.

```
    function test_revokeVesting() public {

        vm.prank(user);

        cl.stake{value: 1000 ether}();

        vm.prank(user2);

        cl.stake{value:1500 ether}();
```

```
        vm.prank(owner);
        cl.closeCrowdloan();
        // 4 Months
        vm.warp(4*30*86400);
        // Add Manager & Revoke Vesting
        vm.startPrank(owner);
        vesting.addManager(owner);
        vesting.revokeVesting(1);
        vm.stopPrank();

        // 7 Months
        vm.warp(6 * 4 weeks + 15 days);
        vm.roll((7 * 4 * 6 + 15) * 7200);

        console.log("User Rewards", cl.userTotalRewards(user));
        console.log("User2 Rewards",cl.userTotalRewards(user2));

        // Claim Rewards User
        vm.prank(user);
        cl.claimRewards();

        // Claim Rewards User2
        vm.prank(user2);
        cl.claimRewards();
    }
```

Impact

The wrong amount of rewards sent, causes Reward Insolvency.

Recommendation

Change the logic behind the rewards calculation.

Status

Resolved

[H-03] LiquidCrowdloan#claimOwnership - Change Ownership of the contract will always set the owner as address 0. This will lead to loss of control of LiquidCrowdloan.

Description

The claimOwnership() function has a problem when someone tries to claim the ownership of the contract after being granted.
This problem will always set newOwner as address 0 and will lead to loss control of the contract.

Vulnerability Details

The problem can be seen in the function below:

```
    function claimOwnership() external {

        if (_grantedOwner != msg.sender) revert NotGrantedOwner();

        _grantedOwner = address(0);

        _transferOwnership(_grantedOwner);

    }
```

As we can see _grantedOwner is set on address 0 and the ownership will be transferred to 0 address.

At this point, even if the Admin, updates the implementation contract of proxy, the owner of the new implementation is still the address 0.

Proof of Concept

An example of a test that simulates this error is shown below.

```
    function test_ownership() public {

        vm.prank(owner);

        cl.grantOwnership(user);

        vm.prank(user);

        cl.claimOwnership();


        // Owner after claim ownership is the 0 address

        console.log("Owner First Implementation", cl.owner());

        clImpl2 = new LiquidCrowdloan();


        // Get First implementation contract

                        address                         firstImpl                       =
admin.getProxyImplementation(ITransparentUpgradeableProxy(address(clProxy)));

        console.log("First Implementation Address", firstImpl);


        //Upgrade prox with new implementation contract

                                bytes               memory              data              =
abi.encodeWithSelector(bytes4(keccak256("upgradeTo(address)")),address(clImpl2));
```

```
        vm.prank(owner);

      admin.upgrade(ITransparentUpgradeableProxy(address(clProxy)), address(clImpl2));



                                            address           secondImpl           =
admin.getProxyImplementation(ITransparentUpgradeableProxy(address(clProxy)));

        console.log("Implementation Upgraded", secondImpl);



        //Owner doesn't change if we create a new implementation contract and is still
the 0 address

        console.log("Owner Second Implementation", cl.owner());

}
```

Impact

Loss of control of liquidCrowdloan contract.

Recommendation

Change the logic of claimOwnership() function, like in the example below:

```
function claimOwnership() external {

        if (_grantedOwner != msg.sender) revert NotGrantedOwner();

        _grantedOwner = msg.sender;

        _transferOwnership(_grantedOwner);

  }
```

Status

Resolved

Medium

[M-01] LiquidCrowdloan#unstake,withdraw - Temporary impossibility for the user, to withdraw ASTR Tokens staked initially.

Description

There is a problem when users try to unstake the ASTR Tokens staked from the LiquidCrowdloan contract, based on time dependency.

When more users try to unstake at the same time, liquidCrowdloan contract can't unstake ASTR tokens from dappsStaking Contract, and withdraw() result impossible.

Vulnerability Details

Let's suppose there are 3 different users: USER, USER2, USER3 that stakes 1000, 1500 and 1001 ASTR Tokens.

After the vesting period, these users are able to unstake their tokens, and withdraw it.

At this point, there is a timing mechanism on the unstake function that is better to check:

```
    function unstake() public whenNotPaused {

        . . .

        _globalUnstake();

        . . .

    }
```

After some operations, `unstake()` function, call the `_globalUnstake()` function:

```
    function _globalUnstake() internal {

        uint256 era = currentEra();

        // checks if enough time has passed

        if (block.number < (lastUnstaked + CHUNK_LEN)) return;

        try

            dappsStaking.unstake(

                IDappsStaking.SmartContract(

                    IDappsStaking.SmartContractType.EVM,

                    abi.encodePacked(liquidStakingAddr)

                ),

                uint128(sumToUnstake)

            )

        . . .

    }
```

If some time has not passed, between an unstake to another, this call doesn't reach the try/catch statement and the execution will stop at `if (block.number < (lastUnstaked + CHUNK_LEN)) return;`

This means that the dappsStaking will not send back the amount of Tokens staked.

At this point, let's suppose this:

```
User staked 1000 ASTR

User2 Staked 1500 ASTR

User3 Staked 1001 ASTR
```

When User calls the first unstake, it goes into the try statement of the _globalUnstake(), and the liquidCrowdloan receives back from DappsStaking, 1000 ASTR.

At the same time, if User2 calls the unstake function at the same moment as User, the try statement in _globalUnstake() will never be reached, because not enough time has passed, and User2 tokens will not be sent back to the liquidCrowdloan contract from the DappsStaking contract.

At this moment if the User2 tries to withdraw() tokens, it will get the error "Not enough Funds" because liquidCrowdloan doesn't have 1500 ASTR tokens, but only 1000 from the first unstake.

At this point, the only way to get back tokens from the stake, is to use the `globalUnstakeAdmin()` function.

This function is callable only from the owner of the contract and will unlock ASTR tokens staked in the DappsStaking.sol.

Proof of Concept

An example of a test that simulates this error is shown below.

```
function test_withdraw() public {

        vm.prank(user);

        cl.stake{value: 1000 ether}();

        vm.prank(user2);

        cl.stake{value:1500 ether}();

        vm.prank(user3);

        cl.stake{value: 1001 ether}();

        vm.prank(owner);

        cl.closeCrowdloan();

        // 7 Months

        vm.warp(7*30*86400);
```

```
        vm.roll((7 * 4 * 6) * 7200);

        // User Unstake

        vm.prank(user);

        cl.unstake();

        // User2 Unstake

        vm.prank(user2);

        cl.unstake();

        // Increase time

        vm.warp(6 * 4 weeks + 15 days);

        vm.roll((7 * 4 * 6 + 15) * 7200);


        vm.prank(user);

        cl.withdraw(0);


        //vm.startPrank(owner);

        //vm.roll(7 * 4 * 6 * 7200 + 8100);

        //cl.globalUnstakeAdmin();

        //vm.stopPrank();


        vm.prank(user2);

        cl.withdraw(0);

    }
```

Impact

Centralization risk, because users have to wait for the call to `globalUnstakeAdmin()` function from the Owner of the contract, to get back their ASTR Tokens.

Recommendation

Change the logic behind the _globalUnstake() function to improve the mechanism of unstake/withdraw functions, and make it less centralised.

Status

Resolved

Low

[L-01] LiquidCrowdloan#stake, becomeRefferer- Frontrunning and possibility of using somebody else referral code.

Description

We have the possibility to frontrun the stake with a referral code of somebody else.

Let's suppose this scenario:

User1 calls the becomeReferrer() function to generate a referralCode for the stake.

Once User1 calls this function, a code is generated (let's say "ABC"), and map ownerToRef is updated.

```
ownerToRef[User1] = "ABC";
```

Now if User2 sees this Ref, can use it to stake some ASTR Tokens, instead of the User2.

Once the code is used from User2, User1 can't use it anymore.

At the same time, User1 can't call the becomeReferrer() function anymore, because it's callable only once from every User.

Recommendation

This part of the stake function should be changed to guarantee that the msg.sender is the real owner of the refCode generated previously:

```
function stake(string memory _ref) public payable notClosed whenNotPaused {

    . . .

    if (keccak256(abi.encodePacked(_ref)) != keccak256(abi.encodePacked(""))) {

        . . .

        if (user == refToOwner[_ref]) revert OwnRefcode();

        . . .

    }

}
```

The correct one:

```
function stake(string memory _ref) public payable notClosed whenNotPaused {

    . . .

    if (keccak256(abi.encodePacked(_ref)) != keccak256(abi.encodePacked(""))) {

        . . .

        if (user != refToOwner[_ref]) revert OwnRefcode();

        . . .

    }

}
```

As you can see the second if statement has been changed from == to !=

Impact

At the moment it doesn't have any relevant effect, because the referral code is not involved in other operations.

ProofOfConcept

```
    function test_frontRunningAndUsingSomebodyElseRefCode() public {

        vm.prank(user);

        string memory ref = cl.becomeReferrer();

        vm.prank(user2);

        cl.stake{value: 1000 ether}(ref);

        vm.prank(user);

        cl.stake{value: 1000 ether}(ref);

    }
```

Status

Acknowledged

[L-02] LiquidCrowdloan#unstake Users are not allowed to unstake a portion of their stake

Description

The function unstake only allows users to unstake their whole staked position. A user can not decide to only unstake a portion of it. This could disincentivize users to stake their funds in the protocol.

the real owner of the refCode generated previously:

```
function unstake() public whenNotPaused {

      uint256 amount = aastr.balanceOf(msg.sender);



      if (amount == 0) revert NotEnoughAASTR();

      if (crowdLoanCloseTime == 0) revert StillOpen();

      if (crowdLoanCloseTime + vestingParams.duration > block.timestamp)

          revert StillLocked();



      uint256 era = currentEra();

      sumToUnstake += amount;

      totalStaked -= amount;



      aastr.burn(msg.sender, amount);



      uint256 lag;

      uint256 currentBlock = block.number;



      if (lastUnstaked + CHUNK_LEN > currentBlock) {

          lag = lastUnstaked + CHUNK_LEN - currentBlock;
```

```
        }

        // create a withdrawal to withdraw unlocked later
        withdrawals[msg.sender].push(
            Withdrawal({val: amount, blockReq: currentBlock, lag: lag})
        );

        _globalUnstake();

        emit Unstake(msg.sender, amount, era);
    }
```

Recommendation

Allow the possibility of unstaking portions of the stake for the users.

Status

Acknowledged

[L-03] ALGMVesting#withdrawStuck - Update references for token interactions as this function consistently interacts with the IALGM Interface. Any modifications to ALGM token functions could result in unforeseen behaviours.

Description

withdrawStuck() function, is used by the Admin of the contract to get out tokens that are stuck in the ALGMVesting contract.

```
function withdrawStuck(...) external onlyRole(MANAGER) {

    if (token == address(algmToken)) {

        uint256 totalBalance = algmToken.balanceOf(address(this));

        ...

        IALGM(token).transfer(to, withdrawQty);

    } else {

        uint256 stuckQty = IALGM(token).balanceOf(address(this));

        ...

        IALGM(token).transfer(to, withdrawQty);

    }

}
```

When Admin tries to call the withdrawStuck(...) function to get back, let's say USDT Tokens from the contract, the else statement, calls the IALGM Interface to send back tokens.

At this moment, ALGM Token contract, inherited from ERC20.sol is a very basic token (no changing for transfer or transferFrom functions), so it works correctly, but it could be better to clarify this point in code and inherit from the base IERC20 interface.

Recommendation

Inherit from IERC20.sol instead of IALGM.sol in the else statement.

Impact

At the moment it doesn't have any relevant effect, because ALGM token is a very basic token, with no modified functions that could lead to unexpected behaviours.

Proof Of Concept

In this moment, is working correctly, as you can see from the Proof Of Concept below:

```
    function test_withdrawStuck() public {

    . . .

    vm.startPrank(user);

    algm.mint(user, 1000 ether);

    algm.transfer(address(vesting), 1000 ether);

    ArthToken = new ArthSwapToken();

    ArthToken.mint(user, 1000 ether);

    ArthToken.toTransferable();

    ArthToken.transfer(address(vesting), 1000 ether);

    vm.stopPrank();


    console.log("ART Token Balance", ArthToken.balanceOf(address(vesting)));

    console.log("ALGM Token Balance", algm.balanceOf(address(vesting)));


    vm.prank(owner);

    vesting.withdrawStuck(address(ArthToken), owner, 1000 ether);
```

```
    console.log("ART Token Balance", ArthToken.balanceOf(address(vesting)));
    console.log("ALGM Token Balance", algm.balanceOf(address(vesting)));
}
```

Status

Resolved

Centralisation

The ALGM 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 ALGM project seems to have a sound and well-tested code base, now that our findings have been resolved or acknowledged to achieve security. Overall, most of the code is correctly ordered and follows industry best practices. The code is well commented as well. To the best of our ability, Hashlock is not able to identify any further vulnerabilities.

Our Methodology

Hashlock strives to maintain a transparent working process and to make our audits a collaborative effort. The objective of our security audits are to improve the quality of systems and upcoming projects we review and to aim for sufficient remediation to help protect users and project leaders. Below is the methodology we use in our security audit process.

Manual Code Review:

In manually analysing all of the code, we seek to find any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behaviour when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our methodologies include manual code analysis, user interface interaction, and 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 unreResolved 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 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.