Smart contract security audit

Atlas Cloud

v.1.0

Table of Contents

1.0 Introduction

1.1 Project engagement

During January of 2022, Atlas Cloud engaged CTDSec to audit smart contracts that they created. The engagement was technical in nature and focused on identifying security flaws in the design and implementation of the contracts. Atlas Cloud provided CTDSec with access to their code repository and whitepaper. Atlas cloud aims to have a community-driven NaaS allowing people to access unparalleled yields using Atlas Nodes.

Atlas architecture is based in:

ATL.sol -> ERC20 token

Atlas token -> Integration of manager and ERC20

Atlas Nodes -> Atlas Node Manager

1.2 Disclaimer

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract, rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that’s able to withstand the network’s fast-paced and rapidly changing environment, we at CTDSec recommend that Atlas Cloud team put in place a bug bounty program to encourage further and active analysis of the smart contract.

2.0 Coverage

2.1 Target Code and Revision

For this audit, we performed research, investigation, and review of the Atlas Cloud contract followed by issue reporting, along with mitigation and remediation instructions outlined in this report. The following code files are considered in-scope for the review:

Source:

contracts-atlascloud.zip [SHA256] -

387a4cde2adcef310c75933e20f648d5087c6997b6b6d22c5474387e91a600cf

About Atlas Cloud:

Atlas is a Node As a Service protocol that allows people to purchase nodes with their own currency (ATLAS). There are two different nodes:

Shared nodes: SD (Shared nodes) are nodes that actively seek out full nodes to help with storage and transactions on the blockchain.

Full dedicated nodes: FDN (Full dedicated nodes) are nodes that are actively storing blockchain data.

It’s important that everyone who decides to use AtlasCloud don’t share their connection credentials to the nodes with anyone.

About the protocol distribution:

60% is used to a distribution pool (rewards to the users).

20% is used for the vault.

20% is used as treasury.

2.2 Attacks made to the contract

In order to check for the security of the contract, we tested several attacks in order to make sure that the contract is secure and follows best practices.

№	Issue description.	Checking status
1	Compiler warnings.	PASSED
2	Race conditions and Reentrancy. Cross-function race conditions.	PASSED
3	Possible delays in data delivery.	PASSED
4	Oracle calls.	PASSED
5	Front running.	LOW ISSUES
6	Timestamp dependence.	PASSED
7	Integer Overflow and Underflow.	PASSED
8	DoS with Revert.	PASSED
9	DoS with block gas limit.	PASSED
10	Methods execution permissions.	PASSED
11	Economy model. If application logic is based on an incorrect economic model, the application would not function correctly and participants would incur financial losses. This type of issue is most often found in bonus rewards systems, Staking and Farming contracts, Vault and Vesting contracts, etc.	PASSED
12	The impact of the exchange rate on the logic.	PASSED
13	Private user data leaks.	PASSED
14	Malicious Event log.	PASSED
15	Scoping and Declarations.	PASSED
16	Uninitialized storage pointers.	PASSED

17	Arithmetic accuracy.	PASSED
18	Design Logic.	LOW ISSUES
19	Cross-function race conditions.	PASSED
20	Safe Zeppelin module.	PASSED
21	Fallback function security.	PASSED
22	Overpowered functions / Owner privileges	MEDIUM ISSUES

3.0 Security Issues

3.1 High severity issues [0]

No high severity issues found.

3.2 Medium severity issues [1]

1. Centralization:

ATL.sol

Owner has authority over the next functions:

UpdateUniswapRouter: Owner can change the uniswap router address.

UpdateContractLock: Owner can update the contract lock.

UpdateAtlasToken: Owner can update the Atlas token.

UpdateDistributionFee: Owner can change the distribution fee (no limits).

UpdatePoolFee: Owner can change the pool fee (no limits).

UpdateTreasuryFee: Owner can change the treasury fee (no limits).

UpdateFuturWall: Owner can change the atlas vault address.

UpdateRewardsWall: Owner can change the reward wallet address.

UpdatePoolWall: Owner can change the pool wallet address.

AtlasNodes.sol:

Owner (Onlysentry) has authority over the next functions:

SetToken: Add the token address.

SetPresaleToken: Add the presale token address.

UpdateNodeCountLimit: Change the value of the node count limit (there is no explicit limit, 0 to infinite).

CreatePresaleNode: Create a presale node adding an address/name to it.

CreateNode: Create a node adding an address/name to it.

cashoutNodeReward: Withdraw node rewards.

devLockNode: Lock a node.

cashoutAllNodeRewards: Withdraw all node rewards.

ChangeNodePrice: Change the price of the node (no limits).

ChangeRewardPerNode: Change the rewards of nodes (no limits).

ChangeClaimTime: Change claim times (no limits).

ChangeGasDistri: Change the gas for the distributions (no limits).

<u>AtlasToken.sol</u>

Owner has authority over the next functions:

UpdatePatchesAdress: Owner can change the patch address.

UpdateUniswapV2Router: Owner can change the uniswap router address.

UpdatePresaleAddress: Owner can change the presale address.

UpdateATLAddress: Owner can change ATL token address.

UpdateSwapTokensAmount: Owner can change swap tokens amount (no limits).

UpdatePassKey: Owner can change the pass key (used to prevent bot attacks).

UpdateBuyerLimitAmount: Owner can change the update buyer limit (no limits).

UpdateContractLock: Owner can change if the contract is locked or not.

UpdateDynamicNodes: Owner can change nodes to dynamic.

UpdateNodeLimitBuyers: Owner can enable/disable limits to the buys of the nodes.

SetAutomatedMarketPair: Owner can add address for AMM pair.

BlackListMaliciousAddress: Owner can blacklist malicious address.

ChangeNodePrice: Owner can change node price (no limits).

ChangeRewardPerNode: Owner can change node rewards (no limits).

ChangeClaimTime: Owner can change claim time (no limits).

ChangeGasDistri: Owner can change gas for distribution (no limits).

PreSaleAtlas.sol

lockUnlockToken: Owner can lock/unlock the token.

setAtlasTokenAddress: Owner can change the atlas token address.

setAtlasVault: Owner can change the atlas vault token address.

sendReversesToDeadWallet: Owner can send tokens to reserve if sale is not completed.

getFTMBalance: Owner can view how much FTM was deposited.

Recommendation:

-Add a time lock on privileged operations.

-Use a multisig wallet to prevent SPOF on the Private Key.

-Introduce DAO mechanism for owner functions (will add transparency and user involvement).

Notes: All team wallets are used with multisig wallets.

3.3 Low severity issues [4]

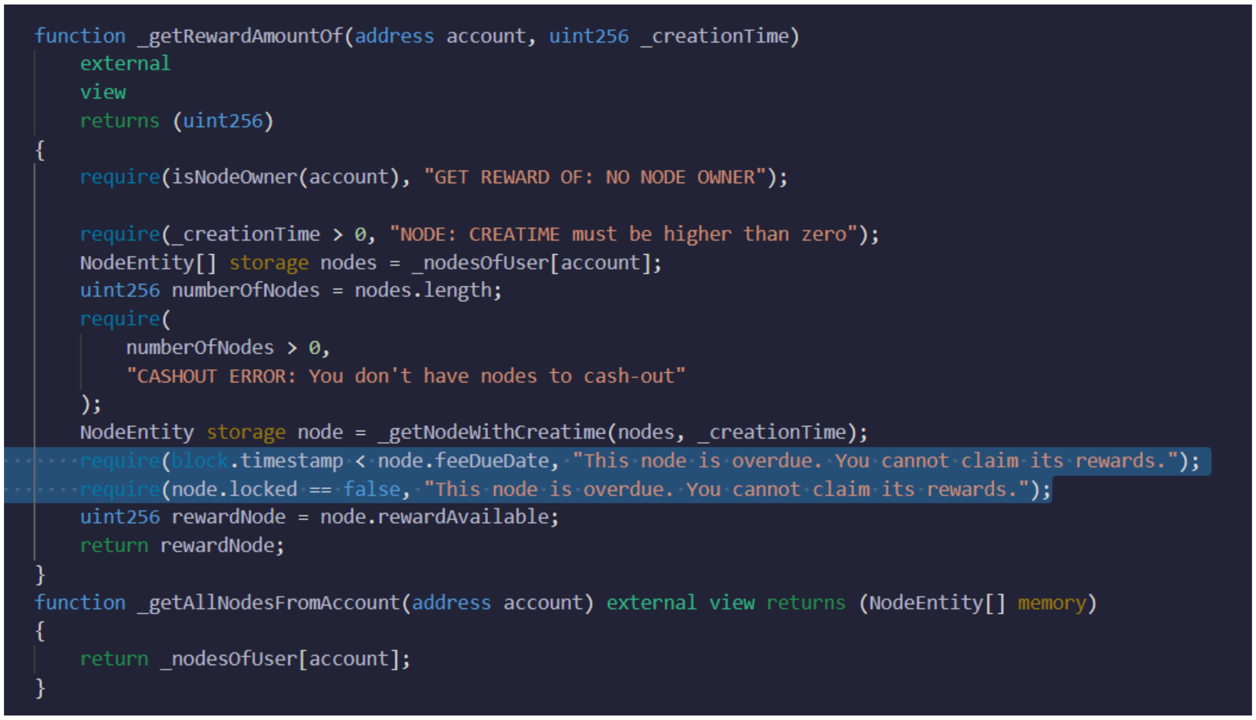
1. Same message in different condition in a require [Atlasnode.sol]

The function '_getRewardAmountOf' have two requires:

- First one will check that the timestamp is lower than the nodefeeduedate, in the case that's not filled the node will be overdue and will not be possible to claim rewards.
- At the second one the require check if the node is not locked, the message returned is the same in the other condition.

Recommendation

We recommend changing the second require to 'This node is locked'.

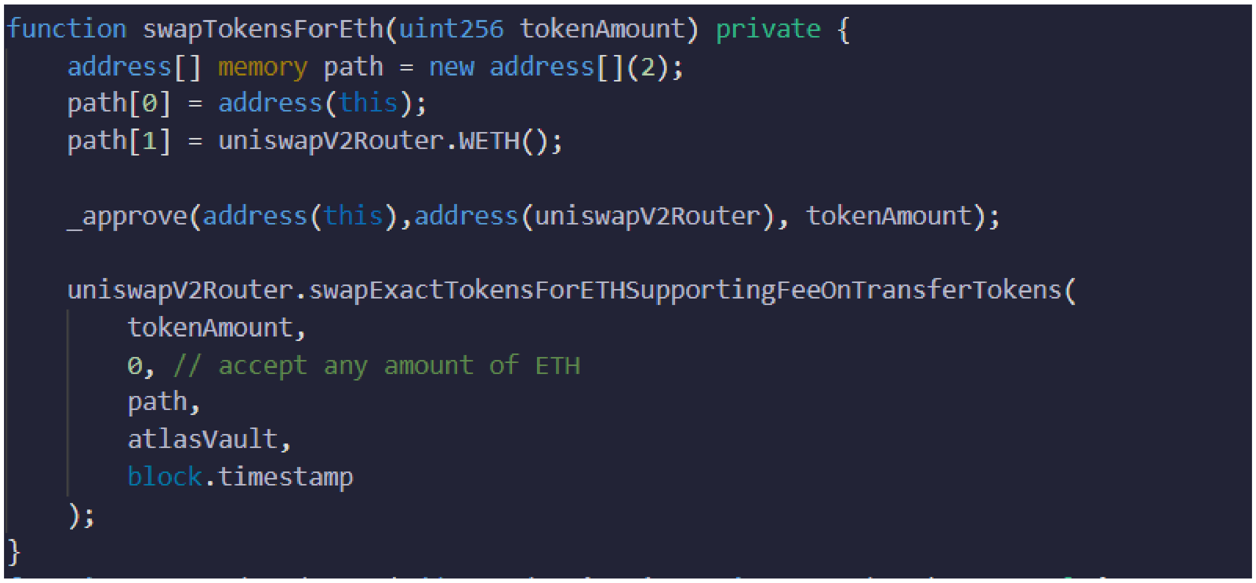

```
function _getRewardAmountOf(address account, uint256 _creationTime)
    external
    view
    returns (uint256)
{
    require(isNodeOwner(account), "GET REWARD OF: NO NODE OWNER");

    require(_creationTime > 0, "NODE: CREATIME must be higher than zero");
    NodeEntity[] storage nodes = _nodesOfUser[account];
    uint256 numberOfNodes = nodes.length;
    require(
        numberOfNodes > 0,
        "CASHOUT ERROR: You don't have nodes to cash-out"
    );
    NodeEntity storage node = _getNodeWithCreatime(nodes, _creationTime);
    require(block.timestamp < node.feeDueDate, "This node is overdue. You cannot claim its rewards.");
    require(node.locked == false, "This node is overdue. You cannot claim its rewards.");
    uint256 rewardNode = node.rewardAvailable;
    return rewardNode;
}
function _getAllNodesFromAccount(address account) external view returns (NodeEntity[] memory)
{
    return _nodesOfUser[account];
}
```

2. Sandwich attack [ATL.sol]

A sandwich attack might happen when an attacker observes a transaction swapping tokens or adding liquidity without setting restrictions on slippage or minimum output amount. The attacker can manipulate the exchange rate by frontrunning (before the transaction being attacked) a transaction to purchase one of the assets and make profits by backrunning (after the transaction being attacked) a transaction to sell the asset, this can happen on large input amounts.

Recommendation:

It's better to set a reasonable minimum output amount, instead of 0 based on token price when you call uniswapV2Router.swapExactTokensForETHSupportingFeeOnTransferTokens.



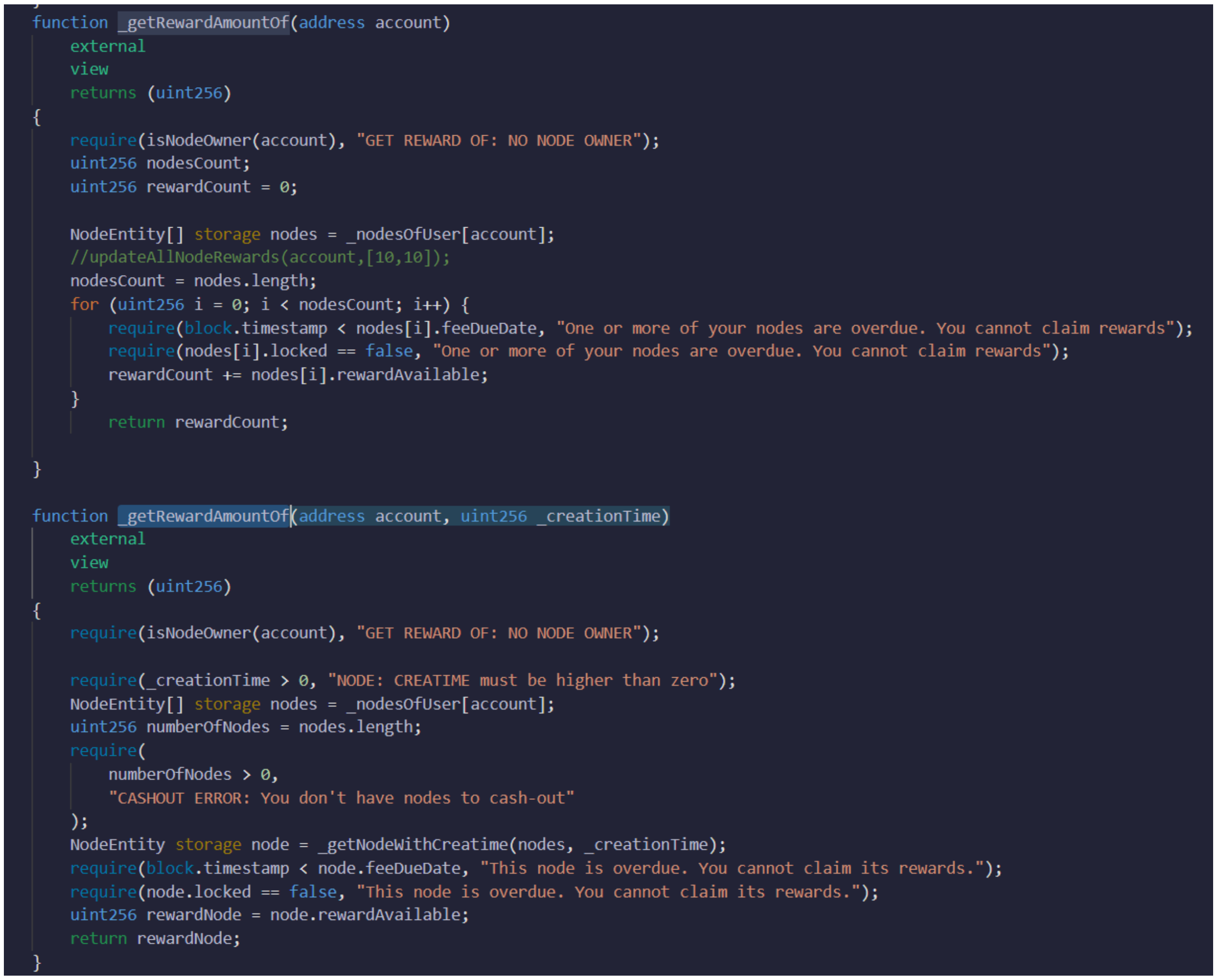
```
function swapTokensForEth(uint256 tokenAmount) private {
    address[] memory path = new address[](2);
    path[0] = address(this);
    path[1] = uniswapV2Router.WETH();

    _approve(address(this),address(uniswapV2Router), tokenAmount);

    uniswapV2Router.swapExactTokensForETHSupportingFeeOnTransferTokens(
        tokenAmount,
        0, // accept any amount of ETH
        path,
        atlasVault,
        block.timestamp
    );
}
```

3. Differentiation of nomenclatures [Atlasnode.sol]

There are view functions with the same name (_getRewardAmountOf).

Recommendation:

We recommend changing the second one to _getRewardAmountCreationTime.



```
function _getRewardAmountOf(address account)
    external
    view
    returns (uint256)
{
    require(isNodeOwner(account), "GET REWARD OF: NO NODE OWNER");
    uint256 nodesCount;
    uint256 rewardCount = 0;

    NodeEntity[] storage nodes = _nodesOfUser[account];
    //updateAllNodeRewards(account,[10,10]);
    nodesCount = nodes.length;
    for (uint256 i = 0; i < nodesCount; i++) {
        require(block.timestamp < nodes[i].feeDueDate, "One or more of your nodes are overdue. You cannot claim rewards");
        require(nodes[i].locked == false, "One or more of your nodes are overdue. You cannot claim rewards");
        rewardCount += nodes[i].rewardAvailable;
    }
        return rewardCount;

}

function _getRewardAmountOf(address account, uint256 _creationTime)
    external
    view
    returns (uint256)
{
    require(isNodeOwner(account), "GET REWARD OF: NO NODE OWNER");

    require(_creationTime > 0, "NODE: CREATIME must be higher than zero");
    NodeEntity[] storage nodes = _nodesOfUser[account];
    uint256 numberOfNodes = nodes.length;
    require(
        numberOfNodes > 0,
        "CASHOUT ERROR: You don't have nodes to cash-out"
    );
    NodeEntity storage node = _getNodeWithCreatime(nodes, _creationTime);
    require(block.timestamp < node.feeDueDate, "This node is overdue. You cannot claim its rewards.");
    require(node.locked == false, "This node is overdue. You cannot claim its rewards.");
    uint256 rewardNode = node.rewardAvailable;
    return rewardNode;
}
```

4. Unused code - Atlastoken.sol

The function 'ChangeGasDistri' is not used/called in the code.

Recommendation

Delete unused code.

4.0 Summary of the audit

The issues found in the audit do not imply a risk for the protocol. It is important to have control over the owner functions, for example, with a delay in their execution & always double-checking the values introduced by the owner in order to avoid errors in the protocol regarding introduction of parameters by the contract owner.