# AquaGoat
# Smart Contract Audit Report

IMMUNE BYTES
Audits

AquaGoat

June 04, 2021

## Introduction

### 1. About AquaGoat

AquaGoat is a decentralized frictionless yield-generation utility eco-token. The token operates on an automated liquidity-locking and self-staking direct distribution protocol, providing safe, secure and hassle-free transactions and yield-generation for all holders.

AquaGoat is the native utility token of the AquaGoat.Finance ecosystem and will be used for:

- Network donation pooling and staking
- E-commerce integration for use in material purchases
- NFT marketplace transactions
- APO-yield farming and staking
- AquaGoat Crypto-Asset exchange

Recognizing the harmful impacts of cryptocurrency mining on the environment, AquaGoat focuses on ecological conservation efforts, helping to offset, mitigate, and potentially reverse the damage done by humans and conventional mining processes of the past, all while generating income for network participants.

### 2. About ImmuneBytes

ImmuneBytes is a security start-up to provide professional services in the blockchain space. The team has hands-on experience in conducting smart contract audits, penetration testing, and security consulting. ImmuneBytes's security auditors have worked on various A-league projects and have a great understanding of DeFi projects like AAVE, Compound, 0x Protocol, Uniswap, dydx.

The team has been able to secure 15+ blockchain projects by providing security services on different frameworks. ImmuneBytes team helps start-up with a detailed analysis of the system ensuring security and managing the overall project.

Visit http://immunebytes.com/ to know more about the services.

## Documentation Details

The AquaGoat team has provided the following doc for the purpose of audit:

1. https://drive.google.com/file/d/1Mtvc579cBt_DE1xOLUHYjuW9ionV5jUR/view

## Audit Process & Methodology

ImmuneBytes team has performed thorough testing of the project starting with analyzing the code design patterns in which we reviewed the smart contract architecture to ensure it is structured and safe use of third-party smart contracts and libraries.

Our team then performed a formal line-by-line inspection of the Smart Contract in order to find any potential issues like Signature Replay Attacks, Unchecked External Calls, External Contract Referencing, Variable Shadowing, Race conditions, Transaction-ordering dependence, timestamp dependence, DoS attacks, and others.

In the Unit testing phase, we run unit tests written by the developer in order to verify the functions work as intended. In Automated Testing, we tested the Smart Contract with our in-house developed tools to identify vulnerabilities and security flaws.

The code was audited by a team of independent auditors which includes -

1. Testing the functionality of the Smart Contract to determine proper logic has been followed throughout.
2. Analyzing the complexity of the code by thorough, manual review of the code, line-by-line.
3. Deploying the code on testnet using multiple clients to run live tests.
4. Analyzing failure preparations to check how the Smart Contract performs in case of bugs and vulnerabilities.
5. Checking whether all the libraries used in the code are on the latest version.
6. Analyzing the security of the on-chain data.

## Audit Details

- Project Name: AquaGoat
- Languages: Solidity(Smart contract)
- Github commit hash/Smart Contract Address for audit:
  - Link: https://bscscan.com/address/0x07af67b392B7A202fAD8E0FBc64C34F33102165B#code
- Rinkeby deployment
  - Aquagoat: 0x6c5B3054847d1a3A4992d940a244e2c16576803B
- Platforms and Tools: Remix IDE, Truffle, Truffle Team, Ganache, Solhint, VScode, Contract Library, Slither, SmartCheck

## Audit Goals

The focus of the audit was to verify that the smart contract system is secure, resilient, and working according to its specifications. The audit activities can be grouped into the following three categories:

1. Security: Identifying security-related issues within each contract and within the system of contracts.
2. Sound Architecture: Evaluation of the architecture of this system through the lens of established smart contract best practices and general software best practices.
3. Code Correctness and Quality: A full review of the contract source code. The primary areas of focus include:
    a. Correctness
    b. Readability
    c. Sections of code with high complexity
    d. Quantity and quality of test coverage

## Security Level References

Every issue in this report was assigned a severity level from the following:

**Admin/Owner Privileges** can be misused either intentionally or unintentionally.

**High severity issues** will bring problems and should be fixed.

**Medium severity issues** could potentially bring problems and should eventually be fixed.

**Low severity issues** are minor details and warnings that can remain unfixed but would be better fixed at some point in the future.

| Issues | High | Medium | Low |
|---|---|---|---|
| Open | - | - | 3 |
| Closed | - | - | - |

## Admin/Owner Privileges

The **admin/owner** of **Aquagoat** smart contract has various privileges over the smart contract. These privileges can be misused either intentionally or unintentionally (in case admin's private key gets hacked). We assume that these extra rights will always be used appropriately. Some of these admin rights are listed below.

1. **In the Aquagoat contract the owner address can exclude any ethereum address from rewards.**
   The **Aquagoat** contract contains an **excludeFromReward()** function by which the **owner** account can exclude any ethereum address from rewards.

2. **In the Aquagoat contract the owner address can include any ethereum address to receive rewards.**
   The **Aquagoat** contract contains an **includeInReward()** function by which the **owner** account can include any ethereum address for rewards.

3. **In the Aquagoat contract the owner address can include and exclude any ethereum address from fee deduction.**
   The **Aquagoat** contract contains **excludeFromFee**() and **includeInFee**() functions by which the **owner** account can include and exclude any ethereum address from fee deductions.

4. **The owner of Aquagoat contract can anytime update the fee, liquidityFee and max allowed transfer amount.**
   The **Aquagoat** contract contains **setTaxFeePercent**(), **setLiquidityFeePercent**() and **setMaxTxPercent**() functions by which the **owner** account can update the **_taxFee**, **_liquidityFee** and **_maxTxAmount** respectively.

5. **The owner of Aquagoat contract can anytime update the *swapAndLiquifyEnabled* status.**
   The **Aquagoat** contract contains the **setSwapAndLiquifyEnabled**() function by which the **owner** account can update the **swapAndLiquifyEnabled** boolean status.

*Recommendation*:
Consider hardcoding predefined ranges or validations for input variables in privileged access functions. Also consider adding some governance for admin rights for smart contracts or use a multi-sig wallet as admin/owner address.

## Low severity issues

1. **Incorrect parameters used in the OwnershipTransferred event.**
   In the **lock**() function of **Ownable** smart contract incorrect parameters are used in emitting the **OwnershipTransferred** event.

   ```
           _owner = address(0);
           _lockTime = now + time;
           emit OwnershipTransferred(_owner, address(0));
   ```

   Here the **_owner** variable used in **OwnershipTransferred** is already set to **zero** address. So it will always seem like the ownership was transferred from zero address to zero address which is incorrect.

   *Recommendation*:
   Consider emitting the event before updating the **_owner** variable.

2. **lock() and unlock() functions can be called several times by the owner.**
   In **Aquagoat** smart contract, after the initial (planned) lock and unlock period these functions can still be called one after the other by the owner.
   Also after the lock period when the contract gets unlocked the **unlock**() function can still be called any number of times by the contract owner. The preconditions in **unlock**() function only checks that the caller is the **_previousOwner** of the contract which is always true.

   *Recommendation*:
   Consider reseting the **_previousOwner** address to address(0) in the **unlock**() function.

3. **Spelling mistake in geUnlockTime() function's name.**
   The **geUnlockTime**() function in **Ownable** smart contract is spelled incorrectly. This function is declared as a public function so it could be used by other smart contract and off-chain scripts.

   *Recommendation*:
   Consider correcting the name of the function.

## Automated Auditing

### Solhint Linting Violations

Solhint is an open-source project for linting solidity code, providing both security and style guide validations. It integrates seamlessly into most mainstream IDEs. We used Solhint as a plugin within our VScode for this analysis. Multiple Linting violations were detected by Solhint, it is recommended to use Solhint's npm package to lint the contracts.

### Contract Library

Contract-library contains the most complete, high-level decompiled representation of all Ethereum smart contracts, with security analysis applied to them in real-time. We performed analysis using contract Library on the Rinkeby address of the Aquagoat contract used during manual testing:

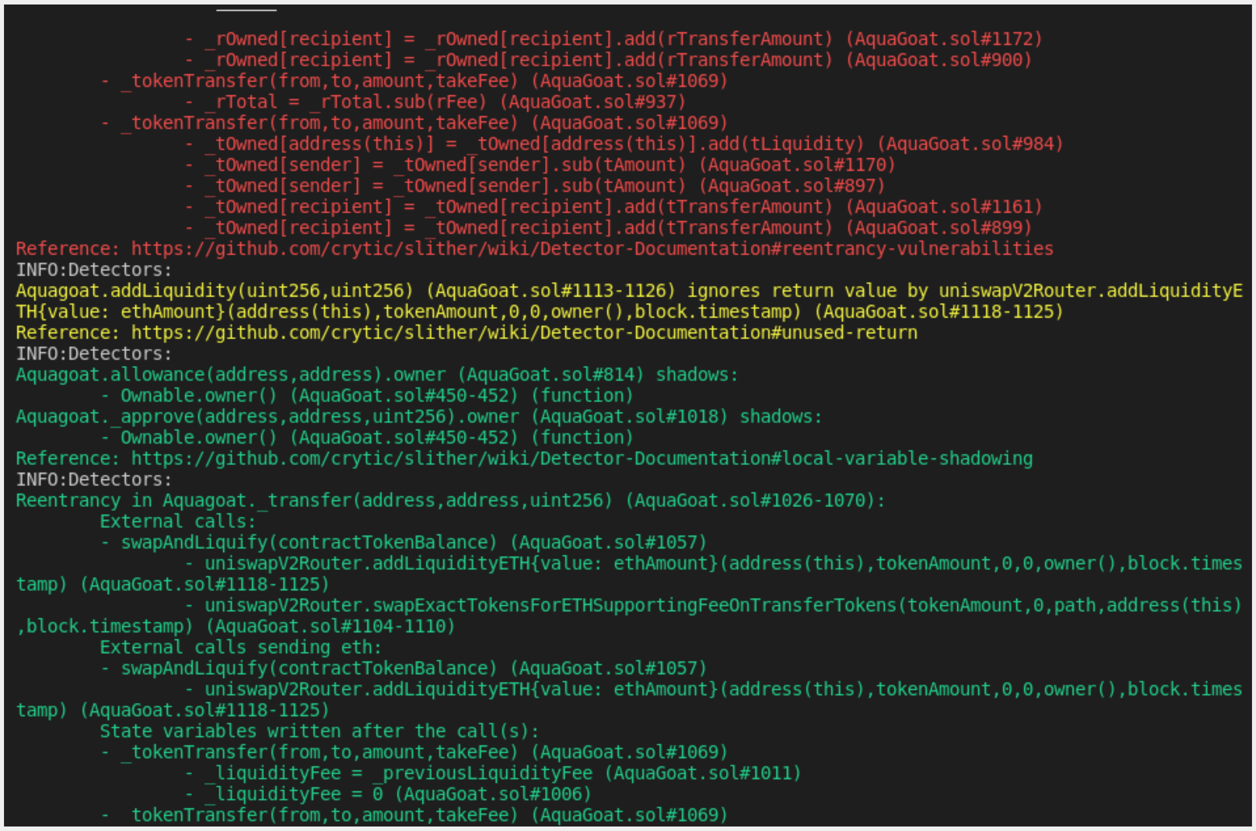
- Aquagoat : 0x6c5B3054847d1a3A4992d940a244e2c16576803B

It raises no major concern for the contracts.

### Slither

Slither, an open-source static analysis framework. This tool provides rich information about Ethereum smart contracts and has critical properties. While Slither is built as a security-oriented static analysis framework, it is also used to enhance the user's understanding of smart contracts, assist in code reviews, and detect missing optimizations.

The concerns slither raises have already been covered in the manual audit section.



```
                - _rOwned[recipient] = _rOwned[recipient].add(rTransferAmount) (AquaGoat.sol#1172)
                - _rOwned[recipient] = _rOwned[recipient].add(rTransferAmount) (AquaGoat.sol#900)
        - _tokenTransfer(from,to,amount,takeFee) (AquaGoat.sol#1069)
                - _rTotal = _rTotal.sub(rFee) (AquaGoat.sol#937)
        - _tokenTransfer(from,to,amount,takeFee) (AquaGoat.sol#1069)
                - _tOwned[address(this)] = _tOwned[address(this)].add(tLiquidity) (AquaGoat.sol#984)
                - _tOwned[sender] = _tOwned[sender].sub(tAmount) (AquaGoat.sol#1170)
                - _tOwned[sender] = _tOwned[sender].sub(tAmount) (AquaGoat.sol#897)
                - _tOwned[recipient] = _tOwned[recipient].add(tTransferAmount) (AquaGoat.sol#1161)
                - _tOwned[recipient] = _tOwned[recipient].add(tTransferAmount) (AquaGoat.sol#899)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities
INFO:Detectors:
Aquagoat.addLiquidity(uint256,uint256) (AquaGoat.sol#1113-1126) ignores return value by uniswapV2Router.addLiquidityE
TH{value: ethAmount}(address(this),tokenAmount,0,0,owner(),block.timestamp) (AquaGoat.sol#1118-1125)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
INFO:Detectors:
Aquagoat.allowance(address,address).owner (AquaGoat.sol#814) shadows:
        - Ownable.owner() (AquaGoat.sol#450-452) (function)
Aquagoat._approve(address,address,uint256).owner (AquaGoat.sol#1018) shadows:
        - Ownable.owner() (AquaGoat.sol#450-452) (function)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#local-variable-shadowing
INFO:Detectors:
Reentrancy in Aquagoat._transfer(address,address,uint256) (AquaGoat.sol#1026-1070):
        External calls:
        - swapAndLiquify(contractTokenBalance) (AquaGoat.sol#1057)
                - uniswapV2Router.addLiquidityETH{value: ethAmount}(address(this),tokenAmount,0,0,owner(),block.times
tamp) (AquaGoat.sol#1118-1125)
                - uniswapV2Router.swapExactTokensForETHSupportingFeeOnTransferTokens(tokenAmount,0,path,address(this)
,block.timestamp) (AquaGoat.sol#1104-1110)
        External calls sending eth:
        - swapAndLiquify(contractTokenBalance) (AquaGoat.sol#1057)
                - uniswapV2Router.addLiquidityETH{value: ethAmount}(address(this),tokenAmount,0,0,owner(),block.times
tamp) (AquaGoat.sol#1118-1125)
        State variables written after the call(s):
        - _tokenTransfer(from,to,amount,takeFee) (AquaGoat.sol#1069)
                - _liquidityFee = _previousLiquidityFee (AquaGoat.sol#1011)
                - _liquidityFee = 0 (AquaGoat.sol#1006)
        - _tokenTransfer(from,to,amount,takeFee) (AquaGoat.sol#1069)
```

## Concluding Remarks

While conducting the audits of AquaGoat smart contract, it was observed that the contract contains only Low severity issues, along with a few areas of Admin/Owner Privileges.

Our auditors suggest that Low severity issues should be resolved by the developers. Resolving the areas of recommendations are up to the team's discretion. The recommendations given will improve the operations of the smart contract.

## Disclaimer

ImmuneBytes's audit does not provide a security or correctness guarantee of the audited smart contract. Securing smart contracts is a multistep process, therefore running a bug bounty program as a complement to this audit is strongly recommended.

Our team does not endorse the AquaGoat platform or its product nor this audit is investment advice.
Notes:

- Please make sure contracts deployed on the mainnet are the ones audited.
- Check for the code refactor by the team on critical issues.

***ImmuneBytes Pvt Ltd.***