BOUNDLESSPAY

Nexus Token

Smart Contract Audit Report

BOUNDLESSPAY

March 10, 2022

Introduction

1. About Boundlesspay

Boundlesspay is a secure application that turns your cell phone into a mobile bank. Its pre-installed digital wallet and debit card enable storing and spending of digital currencies across merchants globally. In addition, you can access crypto loans, pay utilities, invest in digital currencies, and have access to other amazing decentralized finance features in one single app.

Visit https://boundlesspay.com/ to know more about it.

2. About ImmuneBytes

ImmuneBytes is a security start-up to provides 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, and dydx.

The team has been able to secure 105+ blockchain projects by providing security services on different frameworks. ImmuneBytes team helps start-ups 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 Boundlesspay team has provided the following doc for the purpose of audit:

1. https://boundlesspay.com/about/

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: Boudlesspay
- Contracts Name: AntiBotLiquidityGeneratorToken
- Languages: Solidity(Smart contract), Typescript (Unit Testing)
- Token address for initial audit: https://bscscan.com/address/0x693d4af82c298d0b8bd2cdf769e01d8e7d2b6cc3#code%23L1
- 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 Reference

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

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	-	1	4
Closed	-	-	

High Severity Issues

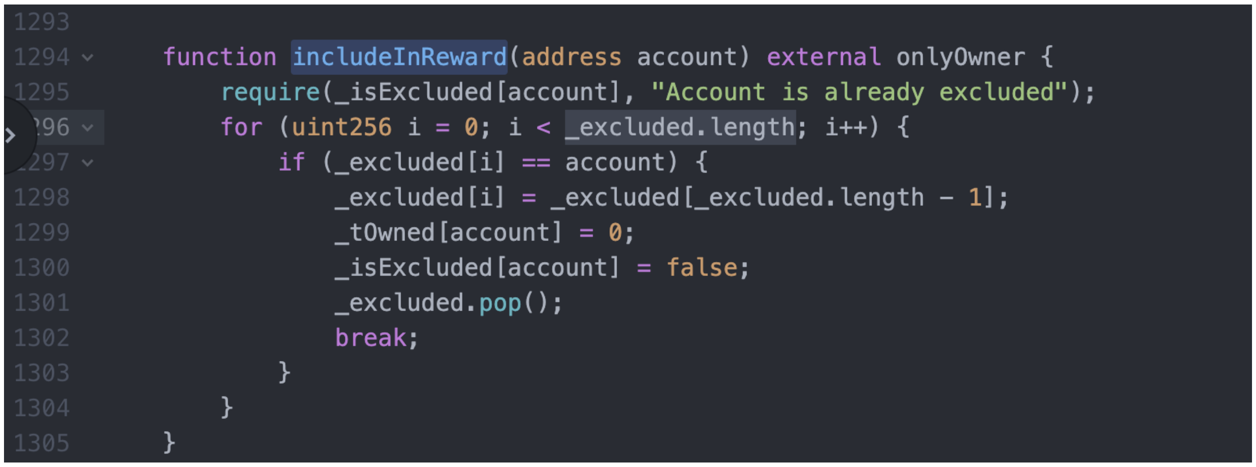
No issues were found.

Medium Severity Issues

1. **Loops are extremely costly**
   Line no -1294-1305

   Description:
   The **AntiBotLiquidityGeneratorToken** contract has some **for loops** in the contract that include state variables like .length of a non-memory array, in the condition of the for loops.

   As a result, these state variables consume a lot more extra gas for every iteration of the for a loop. The following function includes such loops at the above-mentioned lines:
   - **includeInReward**
   - **_getCurrentSupply**



```
1293
1294     function includeInReward(address account) external onlyOwner {
1295         require(_isExcluded[account], "Account is already excluded");
1296         for (uint256 i = 0; i < _excluded.length; i++) {
1297             if (_excluded[i] == account) {
1298                 _excluded[i] = _excluded[_excluded.length - 1];
1299                 _tOwned[account] = 0;
1300                 _isExcluded[account] = false;
1301                 _excluded.pop();
1302                 break;
1303             }
1304         }
1305     }
```

Recommendation:
It's quite effective to use a local variable instead of a state variable like .length in a loop. This will be a significant step in optimizing gas usage.

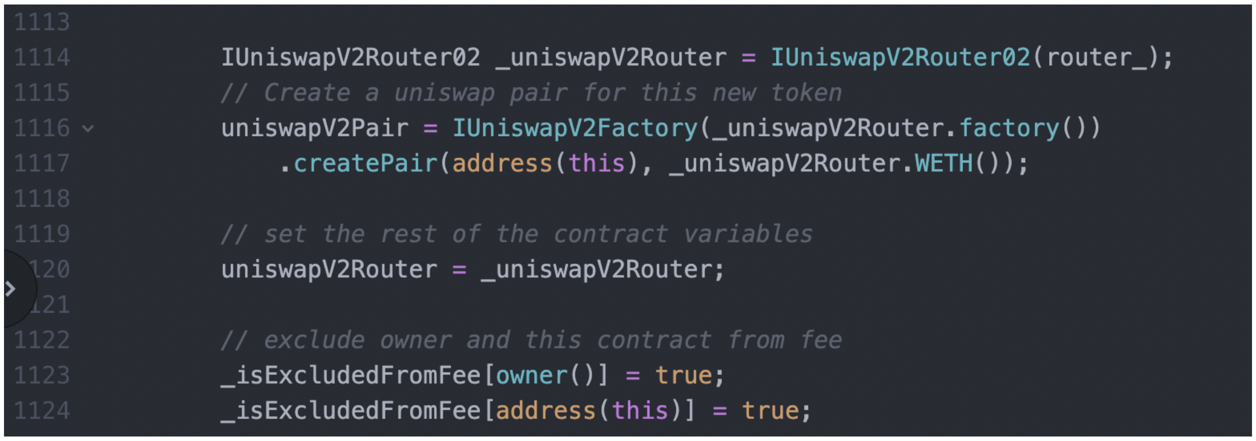
For instance,

```
local_variable = excluded.length;
 for (uint256 index = 0; index < local_variable; index++) {
    if (_rOwned[_excluded[i]] > rSupply || _tOwned[_excluded[i]] > tSupply) return (_rTotal, _tTotal);
          rSupply = rSupply.sub(_rOwned[_excluded[i]]);
          tSupply = tSupply.sub(_tOwned[_excluded[i]]);
    }
}
```

Low Severity Issues

1. **Violation of Check_Effects_Interaction Pattern in the Withdraw function**
   Line no - 1113 - 1124

   Description:
   As per the current design of the **AntiBotLiquidityGeneratorToken** contract, the constructor of the contract includes an external call. However, this external call is made before updating the imperative state variable of the contract.
   This approach violates the Check-Effects Interaction pattern.



```
1113
1114            IUniswapV2Router02 _uniswapV2Router = IUniswapV2Router02(router_);
1115            // Create a uniswap pair for this new token
1116            uniswapV2Pair = IUniswapV2Factory(_uniswapV2Router.factory())
1117                .createPair(address(this), _uniswapV2Router.WETH());
1118
1119            // set the rest of the contract variables
1120            uniswapV2Router = _uniswapV2Router;
1121
1122            // exclude owner and this contract from fee
1123            _isExcludedFromFee[owner()] = true;
1124            _isExcludedFromFee[address(this)] = true;
```

   Therefore, as per the Solidity Guidelines, any modification of the state variables in the base contract must be performed before executing the external call.

   Recommendation:
   Check Effects Interaction Pattern must be followed while implementing external calls in a function.

2. **Absence of Event Emission after imperative State variable Update**

   Description:
   Functions that update an imperative arithmetic state variable contract should emit an event after the update.
   As per the current architecture of the contract, the following functions update crucial arithmetic state variables but don't emit any event on its modification:

   - **setEnableAntiBot()**
   - **setLiquidityFeePercent()**
   - **setTaxFeePercent()**

   The absence of event emission for important state variables update also makes it difficult to track them off-chain as well.

   Recommendation:
   As per the best practices in smart contract development, an event should be fired after changing crucial arithmetic state variables

3. **Constant declaration should be preferred**

   Description:
   State variables that are not supposed to change throughout the contract should be declared as constant.

   Recommendation:
   The following state variables need to be declared as constant unless the current contract design is intended.
   - **_charityAddress**
   - **_decimals**
   - **_name**
   - **_symbol**
   - **_tTotal**

4. **External visibility should be preferred**

   Description:
   Functions that are never called throughout the contract should be marked as **external** visibility instead of **public** visibility.
   This will effectively result in Gas Optimization as well.

Therefore, the following function must be marked as **external** within the contract:

- **isExcludedFromReward()**
- **totalFees()**
- **deliver()**
- **reflectionFromToken()**
- **excludeFromReward()**
- **includeInReward()**
- **excludeFromFee()**
- **includeInFee()**
- **setLiquidityFeePercent()**
- **setSwapAndLiquifyEnabled()**
- **isExcludedFromFee()**

Recommendation:

If the PUBLIC visibility of the above-mentioned functions is not intended, then the EXTERNAL Visibility keyword should be preferred.

Recommendation / Informational

1. **No Setter Function found for the Charity Fee variable**

   During the manual code review it was found that while the contract design includes setter functions for state variables like **_taxFee or _liquidityFee**, it doesn't include any function that allows the owner to update the state of the **_charityFee** state variable.

   Recommendation:
   If the above-mentioned issue is not intended, it is recommended to include a setter for **_charityFee** variable as well if its value is supposed to change in the future. Considering the immutable nature of smart contracts, the owner shall not be able to update the above-mentioned variable as per the current architecture of the contract.

2. **Coding Style Issues in the Contract**

   Description:
   Code readability of a Smart Contract is largely influenced by the Coding Style issues and in some specific scenarios may lead to bugs in the future.

   During the automated testing, it was found that the **AntiBotLiquidityGeneratorToken** contract had quite a few code-style issues.

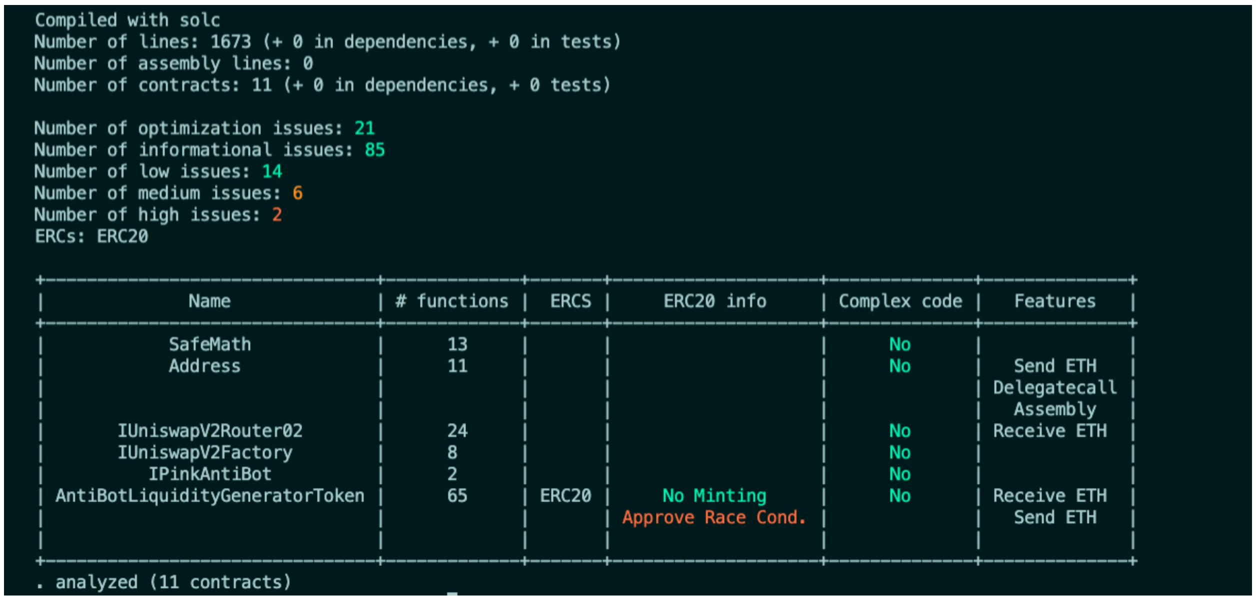
   Recommendation:
   Therefore, it is highly recommended to fix the issues like naming convention, indentation, and code layout issues in a smart contract.

3. **NatSpec Annotations must be included**

   Description:
   The smart contracts do not include the NatSpec annotations adequately.

   Recommendation:
   Cover by NatSpec all Contract methods.

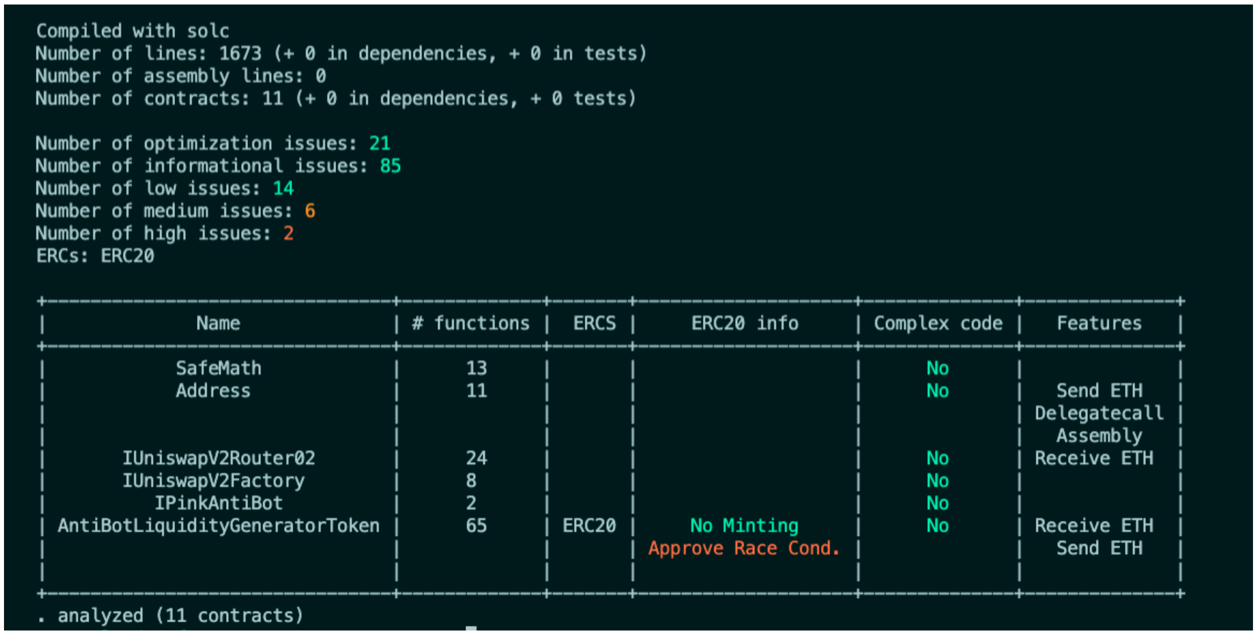
Automated Audit Result



```
Compiled with solc
Number of lines: 1673 (+ 0 in dependencies, + 0 in tests)
Number of assembly lines: 0
Number of contracts: 11 (+ 0 in dependencies, + 0 tests)

Number of optimization issues: 21
Number of informational issues: 85
Number of low issues: 14
Number of medium issues: 6
Number of high issues: 2
ERCs: ERC20

+--------------------------------+-------------+------+--------------------+--------------+--------------+
|              Name              | # functions | ERCS |     ERC20 info     | Complex code |   Features   |
+--------------------------------+-------------+------+--------------------+--------------+--------------+
|            SafeMath            |      13     |      |                    |      No      |              |
|            Address             |      11     |      |                    |      No      |   Send ETH   |
|                                |             |      |                    |              | Delegatecall |
|                                |             |      |                    |              |   Assembly   |
|       IUniswapV2Router02       |      24     |      |                    |      No      | Receive ETH  |
|       IUniswapV2Factory        |      8      |      |                    |      No      |              |
|          IPinkAntiBot          |      2      |      |                    |      No      |              |
| AntiBotLiquidityGeneratorToken |      65     | ERC20|     No Minting     |      No      | Receive ETH  |
|                                |             |      | Approve Race Cond. |              |   Send ETH   |
|                                |             |      |                    |              |              |
+--------------------------------+-------------+------+--------------------+--------------+--------------+
. analyzed (11 contracts)
```



```
Compiled with solc
Number of lines: 1673 (+ 0 in dependencies, + 0 in tests)
Number of assembly lines: 0
Number of contracts: 11 (+ 0 in dependencies, + 0 tests)

Number of optimization issues: 21
Number of informational issues: 85
Number of low issues: 14
Number of medium issues: 6
Number of high issues: 2
ERCs: ERC20

+--------------------------------+-------------+------+--------------------+--------------+--------------+
|              Name              | # functions | ERCS |     ERC20 info     | Complex code |   Features   |
+--------------------------------+-------------+------+--------------------+--------------+--------------+
|            SafeMath            |      13     |      |                    |      No      |              |
|            Address             |      11     |      |                    |      No      |   Send ETH   |
|                                |             |      |                    |              | Delegatecall |
|                                |             |      |                    |              |   Assembly   |
|       IUniswapV2Router02       |      24     |      |                    |      No      | Receive ETH  |
|       IUniswapV2Factory        |      8      |      |                    |      No      |              |
|          IPinkAntiBot          |      2      |      |                    |      No      |              |
| AntiBotLiquidityGeneratorToken |      65     | ERC20|     No Minting     |      No      | Receive ETH  |
|                                |             |      | Approve Race Cond. |              |   Send ETH   |
|                                |             |      |                    |              |              |
+--------------------------------+-------------+------+--------------------+--------------+--------------+
. analyzed (11 contracts)
```

Concluding Remarks

While conducting the audits of the Boundlesspay smart contract, it was observed that the contracts contain Medium and Low severity issues.

Our auditors suggest that Medium and Low severity issues should be resolved by the developers. 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 Boudnlesspay 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