RD AUDITORS

APE TROOP SMART CONTRACT, CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: APE Troop
Prepared on: 19 June 2021
Platform: BSC
Language: Solidity

TABLE OF CONTENTS

Document	4
Introduction	4
Project Scope	5
Executive Summary	6
Code Quality	7
Documentation	8
Use of Dependencies	8
AS-IS Overview	9
Severity Definitions	15
Audit Findings	16
Conclusion	17
Our Methodology	18
Disclaimers	20

THIS DOCUMENT MAY CONTAIN CONFIDENTIAL INFORMATION ABOUT ITS SYSTEMS AND INTELLECTUAL PROPERTY OF THE CUSTOMER AS WELL AS INFORMATION ABOUT POTENTIAL VULNERABILITIES AND METHODS OF THEIR EXPLOITATION.

THE REPORT CONTAINING CONFIDENTIAL INFORMATION CAN BE USED INTERNALLY BY THE CUSTOMER OR IT CAN BE DISCLOSED PUBLICLY AFTER ALL VULNERABILITIES ARE FIXED - UPON DECISION OF CUSTOMER.

Document

Name	**Smart Contract Code Review and Security Analysis Report of ApeTroop**
Platform	BSC / Solidity
File	ApeTroop.sol
MD5 hash	E80A7F7E6E16DFE9363F6C76A5496AB6
SHA256 hash	2953754EBA40E4C8F0E7168C618A30A0A26CBC1B6E4D470274940A355EBAA3EB
Date	19/06/2021

Introduction

RD Auditors (Consultant) were contracted by ApeTroop (Customer) to conduct a Smart Contracts Code Review and Security Analysis. This report represents the findings of the security assessment of the customer`s smart contracts and its code review conducted between 15 -19 June 2021.

This contract consists of one file.

Project Scope

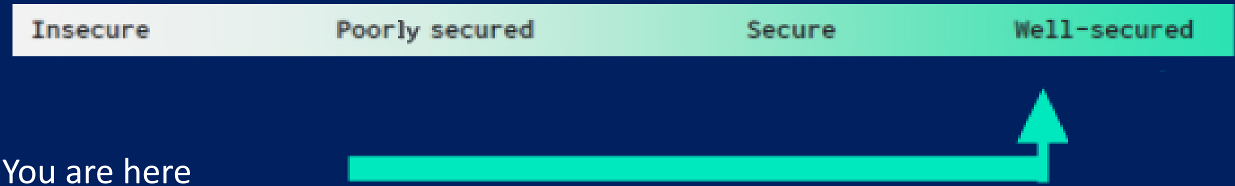
The scope of the project is a smart contract.

We have scanned this smart contract for commonly known and more specific vulnerabilities, below are those considered (the full list includes but is not limited to):

- Reentrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with (Unexpected) Throw
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Byte array vulnerabilities
- Style guide violation
- Transfer forwards all gas
- ERC20 API violation
- Malicious libraries
- Compiler version not fixed
- Unchecked external call - Unchecked math
- Unsafe type inference
- Implicit visibility level

Executive Summary

According to the assessment, the customer's solidity smart contract is well-secured.



Automated checks are with smartDec, Mythril, Slither and remix IDE. All issues were performed by our team, which included the analysis of code functionality, manual audit found during automated analysis were manually reviewed and applicable vulnerabilities are presented in the audit overview section. The general overview is presented in the AS-IS section and all issues found are located in the audit overview section.

We found 0 critical, 0 high, 0 medium, 0 low and 1 very low level issues.

Code Quality

Please find a link that, within this report safeMath, address, ownable and context taken from the popular open source.

The libraries within this smart contract are part of a logical algorithm. A library is a different type of smart contract that contains reusable code. Once deployed on the blockchain (only once), it is assigned to a specific address and its properties/methods can be reused many times by other contracts.

The Ape Troop team has not provided scenario and unit test scripts, which would help to determine the integrity of the code in an automated way.

Overall, the code is not commented. Commenting can provide rich documentation for functions, return variables and more. Use of Ethereum Natural Language Specification Format (NatSpec) for commenting is recommended.

Documentation

We were given the Ape Troop contract as a zip file.

The hash of that file is mentioned in the table. As mentioned above, It's recommended to write comments in the smart contract code, so anyone can quickly understand the programming flow as well as complex code logic.

Comments are very helpful in understanding the overall architecture of the protocol. It also provides a clear overview of the system components, including helpful details, like the lifetime of the background script.

Use of Dependencies

As per our observation, the libraries are used in this smart contract infrastructure. Those were based on well known industry standard open source projects and even core code blocks that are written well and systematically.

AS-IS Overview

Ape Troop

File And Function Level Report

File: ApeTroop.sol

Contract: Context
Observation: Passed
Test Report: Passed
Score: Passed
Conclusion: Passed

Sl.	Function	Type	Observation	Test Report	Conclusion	Score
1	_msgSender	read	Passed	All Passed	No Issue	Passed
2	_msgData	read	Passed	All Passed	No Issue	Passed

Contract: Ownable
Inherit: Context
Observation: Passed
Test Report: Passed
Score: Passed
Conclusion: Passed

Sl.	Function	Type	Observation	Test Report	Conclusion	Score
1	owner	read	Passed	All Passed	No Issue	Passed
2	renounceOwnership	write	Passed	All Passed	No Issue	Passed
3	transferOwnership	write	Passed	All Passed	No Issue	Passed
4	getUnlockTime	read	Passed	All Passed	No Issue	Passed
5	lock	write	Passed	All Passed	No Issue	Passed
6	unlock	write	Passed	All Passed	No Issue	Passed

Contract: Manageable
Inherit: Context
Test Report: Passed
Score: Passed
Conclusion: Passed

Sl.	Function	Type	Observation	Test Report	Conclusion	Score
1	manager	read	Passed	All Passed	No Issue	Passed
2	transferManagement	write	Passed	All Passed	No Issue	Passed
3	SwapExactTokensForETHsupportingFeeOnTransferTokens	write	Passed	All Passed	No Issue	Passed

Contract: Tokenomics
Import: Manageable
Observation: Passed
Test Report: Passed
Score: Passed
Conclusion: Passed

Sl.	Function	Type	Observation	Test Report	Conclusion	Score
1	_addFee	write	Passed	All Passed	No Issue	Passed
2	_addFees	write	Passed	All Passed	No Issue	Passed
3	setMaxTransactionAmount	write	Passed	All Passed	No Issue	Passed
4	_getFeesCount	read	Passed	All Passed	No Issue	Passed
5	_getFeeStruct	read	Passed	All Passed	No Issue	Passed
6	_getFee	read	Passed	All Passed	No Issue	Passed
7	_addFeeCollectedAmount	write	Passed	All Passed	No Issue	Passed
8	getCollectedFeeTotal	read	Passed	All Passed	No Issue	Passed

Contract: PreSaleable
Inherit: Manageable
Observation: Passed
Test Report: Passed
Score: Passed
Conclusion: Passed

Sl.	Function	Type	Observation	Test Report	Conclusion	Score
1	setPreseableEnabled	write	Passed	All Passed	No Issue	Passed

Contract: BaseRfiToken
Inherit: IERC20, IERC20MetaData, Ownable, Presaleable, Tokenomics
Observation: Passed
Test Report: Passed
Score: Passed
Conclusion: Passed

Sl.	Function	Type	Observation	Test Report	Conclusion	Score
1	name	read	Passed	All Passed	No Issue	Passed
2	symbol	read	Passed	All Passed	No Issue	Passed
3	decimals	read	Passed	All Passed	No Issue	Passed
4	totalSupply	read	Passed	All Passed	No Issue	Passed
5	balanceOf	read	Passed	All Passed	No Issue	Passed
6	transfer	write	Passed	All Passed	No Issue	Passed
7	allowance	read	Passed	All Passed	No Issue	Passed
8	approve	write	Passed	All Passed	No Issue	Passed
9	transferFrom	write	Passed	All Passed	No Issue	Passed
10	burn	write	Passed	All Passed	No Issue	Passed
11	_burnTokens	write	Passed	All Passed	No Issue	Passed
12	increaseAllowance	write	Passed	All Passed	No Issue	Passed
13	decreaseAllowance	write	Passed	All Passed	No Issue	Passed
14	isExcludedFromReward	read	Passed	All Passed	No Issue	Passed
15	reflectionFromToken	read	Passed	All Passed	No Issue	Passed
16	tokenFromReflection	read	Passed	All Passed	No Issue	Passed
17	excludeFromReward	write	Passed	All Passed	No Issue	Passed
18	_exclude	write	Passed	All Passed	No Issue	Passed
19	includeInReward	write	Infinite loop possibility	Rectify	Owner must exclude limited wallets	Passed with client consent
20	setExcludedFromFee	write	Passed	All Passed	No Issue	Passed
21	isExcludedFromFee	read	Passed	All Passed	No Issue	Passed
22	_approve	write	Passed	All Passed	No Issue	Passed
23	_isUnlimitedSender	read	Passed	All Passed	No Issue	Passed
24	_isUnlimitedRecipent	read	Passed	All Passed	No Issue	Passed
25	_transfer	write	Passed	All Passed	No Issue	Passed
26	_transferTokens	write	Passed	All Passed	No Issue	Passed
27	_takeFees	write	Passed	All Passed	No Issue	Passed
28	_getValues	read	Passed	All Passed	No Issue	Passed
29	_getCurrentRate	read	Passed	All Passed	No Issue	Passed
30	_getCurrentSupply	read	Passed	All Passed	No Issue	Passed
31	_beforeTokenTransfer	write	Passed	All Passed	No Issue	Passed
32	_getSumOfFees	read	Passed	All Passed	No Issue	Passed
33	_isV2Pair	read	Passed	All Passed	No Issue	Passed
34	_redistribute	write	Passed	All Passed	No Issue	Passed
35	_takeTransactionFees	write	Passed	All Passed	No Issue	Passed

Contract: Liquifier
Import: Manageable, Ownable
Observation: Passed
Test Report: Passed
Score: Passed
Conclusion: Passed

Sl.	Function	Type	Observation	Test Report	Conclusion	Score
1	initializeLiquiditySwapper	write	Passed	All Passed	No Issue	Passed
2	liquify	write	Passed	All Passed	No Issue	Passed
3	_setRouterAddress	write	Passed	All Passed	No Issue	Passed
4	_swapAndLiquify	write	Passed	All Passed	No Issue	Passed
5	_SwapTokensForEth	write	Passed	All Passed	No Issue	Passed
6	_addLiquidity	write	Passed	All Passed	No Issue	Passed
7	_addLiquidity	write	Passed	All Passed	No Issue	Passed
8	setRouterAddress	write	Passed	All Passed	No Issue	Passed
9	setSwapAndLiquifyEnabled	write	Passed	All Passed	No Issue	Passed
10	withdrawlockedEth	write	Passed	All Passed	No Issue	Passed
11	_approveDelegate	write	Passed	All Passed	No Issue	Passed

Contract: AntiWhale
Inherit: Tokenomics
Observation: Passed
Test Report: Passed
Score: Passed
Conclusion: Passed

Sl.	Function	Type	Observation	Test Report	Conclusion	Score
1	_getAntiWhaleFees	read	Passed	All Passed	No Issue	Passed

Contract: SafeToken
Inherit: BaseRfiToken, Liquifier, AntiWhale
Observation: Passed
Test Report: Passed
Score: Passed
Conclusion: Passed

Sl.	Function	Type	Observation	Test Report	Conclusion	Score
1	_isV2Pair	read	Passed	All Passed	No Issue	Passed
2	_getsumOfFees	read	Passed	All Passed	No Issue	Passed
3	_beforeTokenTransfer	write	Passed	All Passed	No Issue	Passed
4	_takeTransactionFees	write	Passed	All Passed	No Issue	Passed
5	_burn	write	Passed	All Passed	No Issue	Passed
6	_takeFee	write	Passed	All Passed	No Issue	Passed
7	_TakeFeeETH	write	Passed	All Passed	No Issue	Passed
8	_approveDelegate	write	Passed	All Passed	No Issue	Passed

Contract: ApeTroop
Inherit: SafeToken
Observation: Passed
Test Report: Passed
Score: Passed
Conclusion: Passed

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to lost tokens etc.
High	High level vulnerabilities are difficult to exploit; however, they also have a significant impact on smart contract execution, e.g. public access to crucial functions.
Medium	Medium level vulnerabilities are important to fix; however, they cannot lead to lost tokens.
Low	Low level vulnerabilities are most related to outdated, unused etc. These code snippets cannot have a significant impact on execution.
Lowest Code Style/ Best Practice	Lowest level vulnerabilities, code style violations and information statements cannot affect smart contract execution and can be ignored.

Audit Findings

Critical

No critical severity vulnerabilities were found.

High

No high severity vulnerabilities were found.

Medium

No medium severity vulnerabilities were found.

Low

No low severity vulnerabilities were found.

Very Low

(1) Infinite loop possibility:

```
516
517     function _getCurrentSupply() internal view returns(uint256, uint256) {
518         uint256 rSupply = _reflectedSupply;
519         uint256 tSupply = TOTAL_SUPPLY;
520         for (uint256 i = 0; i < _excluded.length; i++) {
521             if (_reflectedBalances[_excluded[i]] > rSupply || _balances[_excluded[i]] > tSupply) return (_reflectedSupply, TOTAL_SUPPLY);
522             rSupply = rSupply.sub(_reflectedBalances[_excluded[i]]);
523             tSupply = tSupply.sub(_balances[_excluded[i]]);
524         }
525         if (tSupply == 0 || rSupply < _reflectedSupply.div(TOTAL_SUPPLY)) return (_reflectedSupply, TOTAL_SUPPLY);
526         return (rSupply, tSupply);
527     }
528
```

```
427
428     function includeInReward(address account) external onlyOwner() {
429         require(_isExcludedFromRewards[account], "Account is not excluded");
430         for (uint256 i = 0; i < _excluded.length; i++) {
431             if (_excluded[i] == account) {
432                 _excluded[i] = _excluded[_excluded.length - 1];
433                 _balances[account] = 0;
434                 _isExcludedFromRewards[account] = false;
435                 _excluded.pop();
436                 break;
437             }
438         }
439     }
```

If the _excluded array length is increased, then it might hit the gas limit. Another function also would be affected with this is: _getCurrentSupply but it is a view function so it will not create any serious impact.

Solution: Please keep this excluded wallet to a minimum, ideally under 100 wallets.

Discussion

1. Hardcoded addresses should be double checked.
2. Renounceownership will make all onlyOwner functions inaccessible.

Conclusion

We were given a contract file and have used all possible tests based on the given object. The contract is written systematically, so it is ready to go for production.

Since possible test cases can be unlimited and developer level documentation (code flow diagram with function level description) not provided, for such an extensive smart contract protocol, we provide no such guarantee of future outcomes. We have used all the latest static tools and manual observations to cover maximum possible test cases to scan everything.

The security state of the reviewed contract is "well secured".

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review:

In manually reviewing all of the code, we look for 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 behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use 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 follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever 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 conservative 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, 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 details are made public.

Disclaimers

RD Auditors Disclaimer

The smart contracts given for audit have been analysed in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in 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).

Because the total number of test cases are 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, 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 recommend proceeding with several independent audits and a public bug bounty program to ensure security of smart contracts.

Technical Disclaimer

Smart contracts are deployed and executed on the blockchain. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to hacks. Thus, the audit can't guarantee explicit security of the audited smart contracts.

RD AUDITORS

Email: info@rdauditors.com

Website: www.rdauditors.com