PIG TOKEN PROTOCOL SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer:	Pig Token Team (https://pigtoken.finance)
Prepared on:	24/03/2021
Platform:	Binance Smart Chain
Language:	Solidity
Audit Type:	Standard

audit@etherauthority.io

Table of contents

THIS IS SECURITY AUDIT REPORT DOCUMENT AND WHICH MAY CONTAIN INFORMATION WHICH IS CONFIDENTIAL. WHICH INCLUDES ANY POTENTIAL VULNERABILITIES AND MALICIOUS CODES WHICH CAN BE USED TO EXPLOIT THE SOFTWARE. THIS MUST BE REFERRED INTERNALLY AND ONLY SHOULD BE MADE AVAILABLE TO PUBLIC AFTER ISSUES ARE RESOLVED.

Project file

Name	Smart Contract Code Review and Security Analysis Report for PIG TOKEN
Platform	Binance Smart Chain / Solidity
File 1	PigToken.sol
File 1 MD5 hash	06EC31E96FE528D2CE45BFCD0B55FBEB
File 1 BscScan Contract URL	https://bscscan.com/address/0x8850d2c68c632e3b258e612abaa8fada7e6958e5#code
Date	24/03/2021

Introduction

We were contracted by the Pig Token team to perform the Security audit of the smart contracts code. The audit has been performed using manual analysis as well as using automated software tools. This report presents all the findings regarding the audit performed on 24/03/2021.

The Audit type was Standard Audit. Which means this audit is concluded based on Standard audit scope, which is one security engineer performing audit procedure for 2 days. This document outlines all the findings as well as AS-IS overview of the smart contract codes.

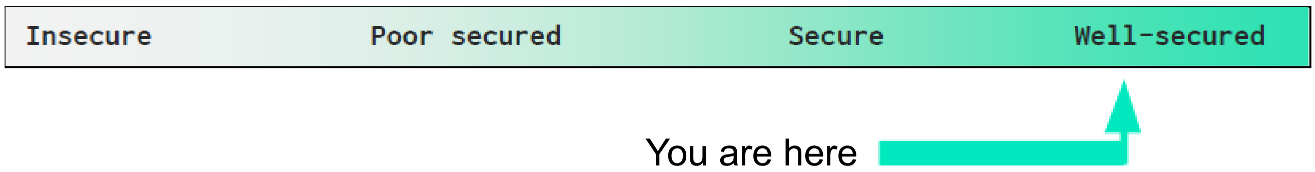
Quick Stats:

Main Category	Subcategory	Result
Contract Programming	Solidity version not specified	Passed
	Solidity version too old	Moderated
	Integer overflow/underflow	Passed
	Function input parameters lack of check	Passed
	Function input parameters check bypass	Passed
	Function access control lacks management	Passed
	Critical operation lacks event log	Passed
	Human/contract checks bypass	Passed
	Random number generation/use vulnerability	N/A
	Fallback function misuse	Passed
	Race condition	Passed
	Logical vulnerability	Passed
	Other programming issues	Passed
Code Specification	Function visibility not explicitly declared	Passed
	Var. storage location not explicitly declared	Passed
	Use keywords/functions to be deprecated	Passed
	Other code specification issues	Passed
Gas Optimization	Assert() misuse	Passed
	High consumption 'for/while' loop	Moderated
	High consumption 'storage' storage	Passed
	"Out of Gas" Attack	Passed
Business Risk	The maximum limit for mintage not set	Passed
	"Short Address" Attack	Passed
	"Double Spend" Attack	Passed

Overall Audit Result: PASSED

Executive Summary

According to the **standard** audit assessment, Customer`s solidity smart contract is **well secured**.



We used various tools like SmartDec, Mythril, Slither and Remix IDE. At the same time this finding is based on critical analysis of the manual audit.
All issues found during automated analysis were manually reviewed and applicable vulnerabilities are presented in the Audit overview section. General overview is presented in AS-IS section and all found issues can be found in the Audit overview section.

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

Code Quality

Pig Token protocol consists of 1 core smart contract file. These smart contracts also contain Libraries, Smart contract inherits and Interfaces. These are compact and well written contracts.

The libraries in the Pig Token protocol are part of its 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 a specific address and its properties / methods can be reused many times by other contracts in the Pig Token protocol.

The Pig Token team has **not** provided scenario and unit test scripts, which would have helped to determine the integrity of the code in an automated way.

Overall, code parts are **not** well commented. Commenting can provide rich documentation for functions, return variables and more. Ethereum Natural Language Specification Format (NatSpec) is recommended.

Documentation

We were given Pig Token smart contracts code in the form of BscScan web link. The hash of that file and its web link are mentioned above in the table.

As mentioned above, most code parts are **not** well commented. so anyone can not quickly understand the programming flow as well as complex code logic. Comments are very helpful in understanding the overall architecture of the protocol.

Another source of information was its official website pigtoken.finance which provided rich information about the project architecture and tokenomics.

Use of Dependencies

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

Apart from libraries, Pig Token smart contract depends on pancakeswap smart contracts as external smart contract calls.

AS-IS overview

Pig Token is a BEP20 standard smart contract. It also has other features like diffraction, swapping, farming, etc. The smart contract was deployed in BSC mainnet at the time of the audit. And ownership was renounced, which is a good thing. Following are the main components of core smart contracts.

PigToken.sol

(1) Interfaces

(a) IERC20: provides BEP20 token standard
(b) IUniswapV2Factory: factory interface of pancakeswap
(c) IUniswapV2Pair: pancakeswap's interface
(d) IUniswapV2Router01: pancakeswap's router 1 interface
(e) IUniswapV2Router02: pancakeswap's router 2 interface

(2) Inherited contracts

(a) Ownable: ownership contract. Ownership is renounced
(b) Context: provides msg.sender and msg.value context
(c) IERC20: provides BEP20 functions

(3) Usages

(a) using SafeMath for uint256
(b) using Address for address

(4) Events

(a) event MinTokensBeforeSwapUpdated(uint256 minTokensBeforeSwap);
(b) event SwapAndLiquifyEnabledUpdated(bool enabled);
(c) event SwapAndLiquify(uint256 tokensSwapped, uint256 ethReceived, uint256 tokensIntoLiqudity);

(5) Functions

Sl.	Function	Type	Observation	Conclusion	Score
1	constructor	write	Passed	No Issue	Passed
2	name	read	Passed	No Issue	Passed
3	symbol	read	Passed	No Issue	Passed
4	decimals	read	Passed	No Issue	Passed
5	totalSupply	read	Passed	No Issue	Passed
6	balanceOf	read	Passed	No Issue	Passed
7	transfer	write	Passed	No Issue	Passed
8	allowance	read	Passed	No Issue	Passed
9	approve	write	Passed	No Issue	Passed
11	transferFrom	write	Passed	No Issue	Passed
12	increaseAllowance	write	Passed	No Issue	Passed
13	decreaseAllowance	write	Passed	No Issue	Passed
14	isExcludedFromReward	read	Passed	No Issue	Passed
15	totalFees	read	Passed	No Issue	Passed
16	deliver	write	Passed	No Issue	Passed
17	reflectionFromToken	read	Passed	No Issue	Passed
18	tokenFromReflection	read	Passed	No Issue	Passed
19	excludeFromReward	write	Renounced	No Issue	Passed
20	includeInReward	write	Renounced	No Issue	Passed
21	_transferBothExcluded	private	Passed	No Issue	Passed
22	includeInFee	write	Renounced	No Issue	Passed
23	setTaxFeePercent	write	Renounced	No Issue	Passed
24	setLiquidityFeePercent	write	Renounced	No Issue	Passed
25	setMaxTxPercent	write	Renounced	No Issue	Passed
26	setSwapAndLiquifyEnabled	write	Renounced	No Issue	Passed
27	_reflectFee	private	Passed	No Issue	Passed
28	_getValues	private	Passed	No Issue	Passed
29	_getTValues	private	Passed	No Issue	Passed
30	_getRValues	private	Passed	No Issue	Passed
31	_getRate	private	Passed	No Issue	Passed
32	_getCurrentSupply	private	Passed	No Issue	Passed
33	_takeLiquidity	private	Passed	No Issue	Passed
34	calculateTaxFee	private	Passed	No Issue	Passed
35	calculateLiquidityFee	private	Passed	No Issue	Passed
36	removeAllFee	private	Passed	No Issue	Passed
37	restoreAllFee	private	Passed	No Issue	Passed
38	isExcludedFromFee	read	Passed	No Issue	Passed

39	_approve	private	Passed	No Issue	Passed
40	_transfer	private	Passed	No Issue	Passed
41	swapAndLiquify	private	Passed	No Issue	Passed
42	swapTokensForEth	private	Passed	No Issue	Passed
43	addLiquidity	private	Passed	No Issue	Passed
44	_tokenTransfer	private	Passed	No Issue	Passed
45	_transferStandard	private	Passed	No Issue	Passed
46	_transferToExcluded	private	Passed	No Issue	Passed
47	_transferFromExcluded	private	Passed	No Issue	Passed

Severity Definitions

Risk Level	**Description**
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to tokens loss etc.
High	High-level vulnerabilities are difficult to exploit; however, they also have significant impact on smart contract execution, e.g. public access to crucial functions
Medium	Medium-level vulnerabilities are important to fix; however, they can't lead to tokens lose
Low	Low-level vulnerabilities are mostly related to outdated, unused etc. code snippets, that can't have significant impact on execution
Lowest / Code Style / Best Practice	Lowest-level vulnerabilities, code style violations and info statements can't affect smart contract execution and can be ignored.

Audit Findings

Critical

No critical severity vulnerabilities were found.

High

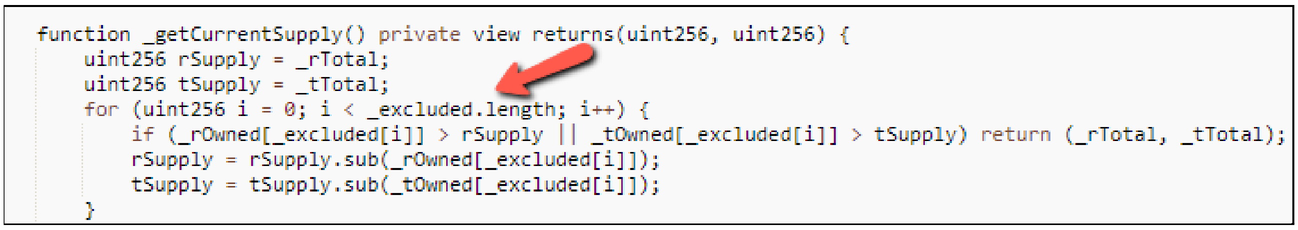
No high severity vulnerabilities were found.

Medium

No medium severity vulnerabilities were found.

Low

(1) Infinite loops possibility at two places:



```
function _getCurrentSupply() private view returns(uint256, uint256) {
    uint256 rSupply = _rTotal;
    uint256 tSupply = _tTotal;
    for (uint256 i = 0; i < _excluded.length; i++) {
        if (_rOwned[_excluded[i]] > rSupply || _tOwned[_excluded[i]] > tSupply) return (_rTotal, _tTotal);
        rSupply = rSupply.sub(_rOwned[_excluded[i]]);
        tSupply = tSupply.sub(_tOwned[_excluded[i]]);
    }
```

includeInReward and _getCurrentSupply have a loop without explicit limits in _excluded array.

Resolution: Since ownership is renounced, the owner can not exclude any more wallets and this issue is **auto-resolved now**.

Very Low

(1) Use the latest solidity version while contract deployment to prevent any compiler version level bugs.

Resolution: This issue is acknowledged.

(3) Event log must be fired in place where the stats are being changed. for example:

- deliver
- excludeFromReward
- includeInReward

Resolution: This issue is acknowledged.

Discussion / Best practices:

(1) Approve of BEP20 standard: This can be used to front run. From the client side, only use this function to change the allowed amount to 0 or from 0 (wait till transaction is mined and approved). This should be done from the client side.

(2) All functions which are not called internally, must be declared as external. It is more efficient as sometimes it saves some gas.
https://ethereum.stackexchange.com/questions/19380/external-vs-public-best-practices

Conclusion

We were given contract code. And we have used all possible tests based on given objects as files. The contracts are written so systematically, that we did not find any major issues. **So it is good to go for the production.**

Since possible test cases can be unlimited for such extensive smart contract protocol, so 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.

Smart contracts within the scope were manually reviewed and analyzed with static analysis tools. Smart Contract’s high level description of functionality was presented in As-is overview section of the report.

Audit report contains all found security vulnerabilities and other issues in the reviewed code.

Security state of the reviewed contract, based on extensive audit procedure scope 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 analyze 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 scrutinized 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

EtherAuthority.io Disclaimer

EtherAuthority team has analyzed this smart contract 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).

Due to the fact that the total number of test cases are unlimited, so 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 to conduct a bug bounty program to confirm the high level of security of this smart contract.

Technical Disclaimer

Smart contracts are deployed and executed on blockchain platform. 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.

EtherAuthority