RD AUDITORS

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer:	Wault
Prepared on:	10/03/2021
Platform:	Binance Smart Chain
Language:	Solidity

info@rdauditors.com

Table of contents

Document

Name	Smart Contract Code Review and Security Analysis Report for Wault
Platform	BSC / Solidity
File 1	WaultLocker.sol
MD5 hash	DC12CAE26604D837539B68448CF5D58B
SHA256 hash	818F2B6E7FFC66C57215393221F4E211DECC89EA5F1D007037E996FBA87CA4B7
Date	**10/03/2021**

Introduction

RD Auditors (Consultant) was contracted by Wault Team (Customer) to conduct a Smart Contracts Code Review and Security Analysis. This report presents the findings of the security assessment of Customer`s smart contracts and its code review conducted between March 8, 2021 – March 10, 2021.

This contract consists of 1 file.

Project Scope

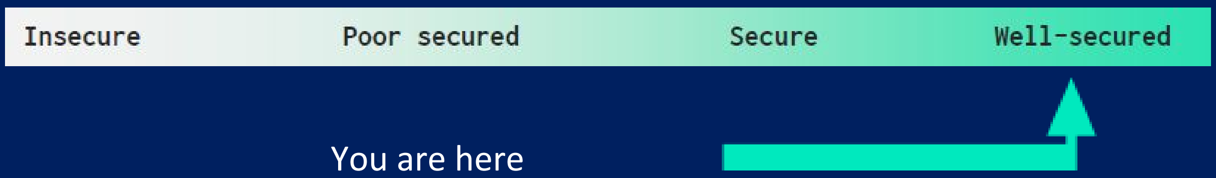
The scope of the project is a smart contract.

We have scanned this smart contract for commonly known and more specific vulnerabilities, below are those considered (the full list includes but 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 2 critical, 0 high, 0 medium, 0 low and 1 very low level issues.

UPDATE(11/03/2021): After modification of code we found 0 critical, 0 high, 0 medium, 0 low and 0 very low level issues

Code Quality

Wault protocol consists of a single smart contract file. Wault team has also conducted unit tests using script provided through the same github link which fortify functionality and security of the contract, which also helped us to determine the integrity of the code in an automated way.

Overall, the code is not commented. Commenting provides rich documentation for functions, return variables and more and also helps auditors to quick cover the flow behind code logic. Use of Ethereum Natural Language Specification Format (NatSpec) for commenting is recommended.

Documentation

We were given a WaultLock contract and its supporting files in the form of a github link:
https://github.com/WaultFinance/WAULT/blob/master/contracts/WaultLocker.sol
The hash of that file is mentioned in the table. As mentioned, 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

Core code blocks are written well and systematically. No other dependencies except safeMath, Ownable, IERC20.

AS-IS overview

WaultLock contract overview

WaultLock is a token locker contract.

File And Function Level Report

File: WaultLock.sol

Contract: WaultLocker
Import:
"https://github.com/OpenZeppelin/openzeppelin-contracts/blob/release-v3.4/contracts/token/ERC20/IERC20.sol";
"https://github.com/OpenZeppelin/openzeppelin-contracts/blob/release-v3.4/contracts/math/SafeMath.sol";
"https://github.com/OpenZeppelin/openzeppelin-contracts/blob/release-v3.4/contracts/access/Ownable.sol";
Observation: Not Passed
Test Report: Not Passed
Score: passed
Conclusion: passed

Sl.	Function	Type	Observation	Test Report	Conclusion	Score
1	LockToken	write	Passed	All Passed	No Issue	Passed
2	addCustomLock	write	Passed	All Passed	No Issue	Passed
3	CustomLockTokens	write	passed	All Passed	No Issue	Passed
4	WithdrawTokens	write	passed	All passed	No Issue	Passed
5	SetWaultMarking Address	read	passed	All Passed	No Issue	passed
6	getDepositsByWithdrawal Address	read	Passed	All Passed	No Issue	Passed
7	getDepositsByTokenAddress	read	Passed	All Passed	No Issue	Passed
8	getTokenTotalLockedBalance	read	Passed	All Passed	No Issue	Passed

UPDATE(10/03/2021): After our suggestions the code of function "LockToken" and "WithdrawTokens" has been modified and thus does not have any vulnerability.

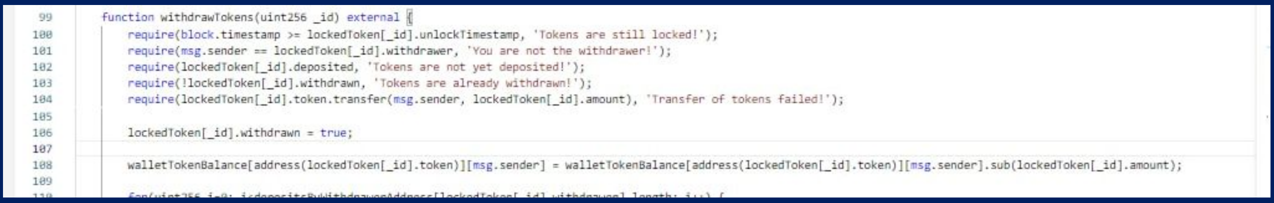
Severity Definitions

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

1. In the "withdrawToken" function there is a probability of reentrancy before variable update.

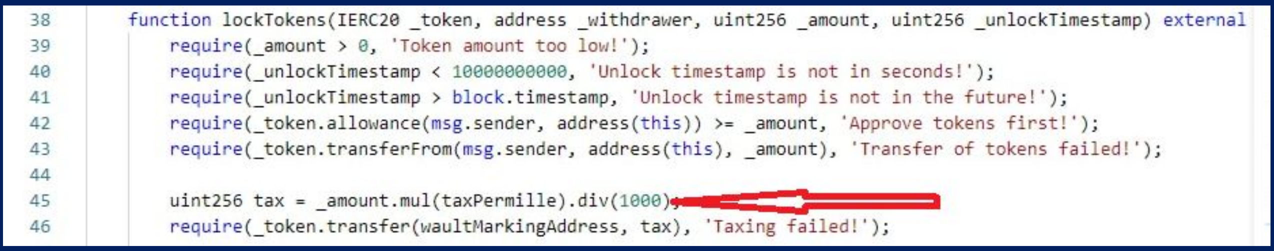


```
function withdrawTokens(uint256 _id) external {
    require(block.timestamp >= lockedToken[_id].unlockTimestamp, 'Tokens are still locked!');
    require(msg.sender == lockedToken[_id].withdrawer, 'You are not the withdrawer!');
    require(lockedToken[_id].deposited, 'Tokens are not yet deposited!');
    require(!lockedToken[_id].withdrawn, 'Tokens are already withdrawn!');
    require(lockedToken[_id].token.transfer(msg.sender, lockedToken[_id].amount), 'Transfer of tokens failed!');

    lockedToken[_id].withdrawn = true;

    walletTokenBalance[address(lockedToken[_id].token)][msg.sender] = walletTokenBalance[address(lockedToken[_id].token)][msg.sender].sub(lockedToken[_id].amount);

```

Solution: transfer should be on the last line to be on the safe side for high frequency calling from the same address.

UPDATE(11/03/2021): After modification of code there is no shortage here

2. In "LockToken"function Minimum deposit amount should be a certain digit while tax calculation, if deposit amount is less than 500 than tax will be 0, as figure applied in contract.



```
function lockTokens(IERC20 _token, address _withdrawer, uint256 _amount, uint256 _unlockTimestamp) external
    require(_amount > 0, 'Token amount too low!');
    require(_unlockTimestamp < 10000000000, 'Unlock timestamp is not in seconds!');
    require(_unlockTimestamp > block.timestamp, 'Unlock timestamp is not in the future!');
    require(_token.allowance(msg.sender, address(this)) >= _amount, 'Approve tokens first!');
    require(_token.transferFrom(msg.sender, address(this), _amount), 'Transfer of tokens failed!');

    uint256 tax = _amount.mul(taxPermille).div(1000);
    require(_token.transfer(waultMarkingAddress, tax), 'Taxing failed!');
```

Solution: minimum deposit amount should be a certain digit while tax calculation.

UPDATE(11/03/2021): After our suggestion the code has been modified and all the deficiencies have been removed.

High

No high severity vulnerabilities were found.

Medium

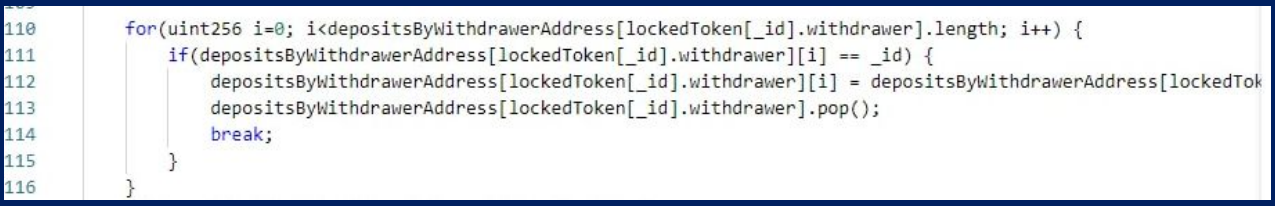
No Medium severity vulnerabilities were found.

Low

No Low severity vulnerabilities were found.

Very Low

1. In "withdrawTokens" function depositors may intentionally exploit the system by increasing the loop count by a small amount.



```
        for(uint256 i=0; i<depositsByWithdrawerAddress[lockedToken[_id].withdrawer].length; i++) {
            if(depositsByWithdrawerAddress[lockedToken[_id].withdrawer][i] == _id) {
                depositsByWithdrawerAddress[lockedToken[_id].withdrawer][i] = depositsByWithdrawerAddress[lockedTok
                depositsByWithdrawerAddress[lockedToken[_id].withdrawer].pop();
                break;
            }
        }
```

Solution: loop should be limited for max iteration.

UPDATE(11/03/2021) : Now the loop has been removed and code has been managed without a loop.

Conclusion

We were given a contract file. And we have used all possible tests based on the given object. The contract is written systematically but comments were missing. We found no critical issues So it is good 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 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.

Security state of 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 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

RD Auditors Disclaimer

The smart contracts given for audit have been analyzed 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, 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 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 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.

RD
AUDITORS

info@rdauditors.com