ARBITECH
SOLUTIONS

DOGECASH ERC20 TOKEN SMART CONTRACT CODE REVIEWAND SECURITY ANALYSIS REPORT

Customer: DGC Team
Prepared on: 09/07/2021
Platform: ETHERSCAN
Language: Solidity
Audit Type: Standard

Table of contents

Documents

Name	Code Review and Security Analysis Report for Dogecash ERC20 Token Smart Contract
Platform	ETHERSCAN / Solidity
File name1	Dogecash.sol
SHA256 hash	0xa29c181dd09e75059a932fbd887281476c603c609984308299f2657f3f617d48
Initial Audit Date	09/07/2021

Introduction

We were contracted by the DGC team to perform the Security audit of the Dogecash ERC20 Token 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 09/07/2021.

Audit type was Standard Audit. Which means one senior auditor performing an audit for 2 days. So, this audit is concluded based on standard audit scope. And because the use case scenarios are unlimited, it is encouraged to perform an Extensive audit (which is performed by 2 or more auditors for about 4 days) to come to a more solid conclusion.

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	N/A
	Race condition	Passed
	Logical vulnerability	Passed
	Other programming issues	Passed
Code Specification	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	Passed
	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. Again, it is recommended to perform an Extensive audit assessment to bring a more assured conclusion.

| Insecure | Poor secured | Secure | Well-secured |

you are here
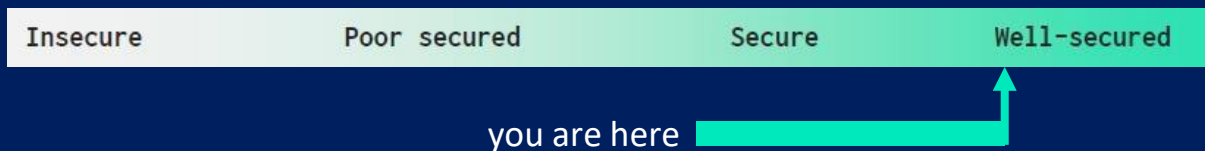
We used various tools like Mythril, Slither and Remix IDE. At the same time this finding is based on critical analysis of the manual audit.

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 critical, 0 high, 0 medium and 1 low level issues.

Code Quality

The Dogecash ERC20 Token protocol consists of one smart contract. It has other inherited contracts like Percentage and Divide. These are compact and wellwritten contracts.

Libraries used in Dogecash ERC20 Token are part of its logical algorithm. They are smart contracts which contain 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 protocol.

The DGC 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.

Documentation

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. We were given a Dogecash ERC20 Token smart contract code in the form of File. The hash of that code is mentioned above in the table.

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 even core code blocks are written well and systematically. This smart contract does not interact with other external smart contracts.

AS-IS overview

It is an ERC20 Token Smart Contract Running on ETHERSCAN Blockchain.

(1) Libraries

  (a) using Safe Math for uint256

(2) Events

  (a) event FrozenFunds(address target, bool frozen) ;
  (b) event TimeLockFunds(address target, uint releasetime);
  (c) event Transfer(address indexed from, address indexed to, uint256 value);
  (d) event Burn(address indexed burner, uint256 value);
  (e) event Approval(address indexed owner, address indexed spender, uint256 value);

(3) Functions

Sl.	Function	Type	Observation	Conclusion	Score
1	addDistributionContract	Write	Passed	No Issue	Passed
2	approve	Write	Passed	No Issue	Passed
3	burn	Write	Passed	No Issue	Passed
4	decreaseApproval	Write	Passed	No Issue	Passed
5	freezeAccount	Write	Admin function	No Issue	Passed
6	increaseApproval	Write	Passed	No Issue	Passed
7	Pause	Write	Admin function	No Issue	Passed
8	timelockAccount	Write	Admin function	No Issue	Passed
9	timeunlockAccount	Write	Admin function	No Issue	Passed
10	transfer	Write	Passed	No Issue	Passed
11	transferFrom	Write	Passed	No Issue	Passed
12	transferOwnership	Write	Admin function	No Issue	Passed
13	unFreezeAccount	Write	Admin function	No Issue	Passed
14	unpause	Write	Admin function	No Issue	Passed

15	allowance	Read	Passed	No Issue	Passed
16	balanceOf	Read	Passed	No Issue	Passed
17	frozen	Read	Passed	No Issue	Passed
18	function releasetime	Read	Passed	No Issue	Passed
19	totalSupply	Read	Passed	No Issue	Passed

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to token 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) Compiler version can be upgraded.

```
pragma solidity ^0.5.16;
```

Although this does not raise any security vulnerability, using the latest compiler version can help to prevent any compiler level bugs.
Solution: This issue is acknowledged.

Conclusion

We were given a contract code. And we have used all possible tests based on given objects as files. So it is good to go for production.

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

Privacy Arbitech Solutions Disclaimer

Arbitech Solutions 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, 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, bug free 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 conducting a bug bounty program to confirm the high level of security of this smart contract.

Technical Disclaimer

Smart contracts are deployed and executed on the 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.

ARBITECH
SOLUTIONS