SMART CONTRACT AUDIT REPORT

for

DOGOToken

Prepared By: Yiqun Chen

Hangzhou, China

August 29, 2021

Document Properties

Client	FROZEN LTD.
Title	Smart Contract Audit Report
Target	DOGOToken
Version	1.0
Author	Shulin Bie
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 29, 2021	Shulin Bie	Final Release

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction 4
- 1.1 About DOGOToken . . . 4
- 1.2 About PeckShield . . . 5
- 1.3 Methodology . . . 5
- 1.4 Disclaimer . . . 7

2 Findings 9
- 2.1 Summary . . . 9
- 2.2 Key Findings . . . 10

3 Detailed Results 11
- 3.1 Potential Sandwich/MEV Attack In DOGOToken . . . 11
- 3.2 Improved Validation Of Function Arguments . . . 12
- 3.3 Trust Issue Of Admin Keys . . . 13
- 3.4 Potential Lockup Of ETH Leftover In swapAndLiquify() . . . 14
- 3.5 Inconsistent Logic Between State Initialization And Their Updates . . . 16

4 Conclusion 19

References 20

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the DOGOToken protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About DOGOToken

The DOGOToken is widely used in the DogemonGo game, which is a completely free crypto game that makes use of Augmented Reality (AR) to deliver an interactive experience. The DogemonGo game is an unique combination of game, meme and crypto-culture, which enriches the DeFi ecosystem and also presents a unique contribution to current DeFi ecosystem.

The basic information of DOGOToken is as follows:

Table 1.1: Basic Information of DOGOToken

Item	Description
Target	DOGOToken
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 29, 2021

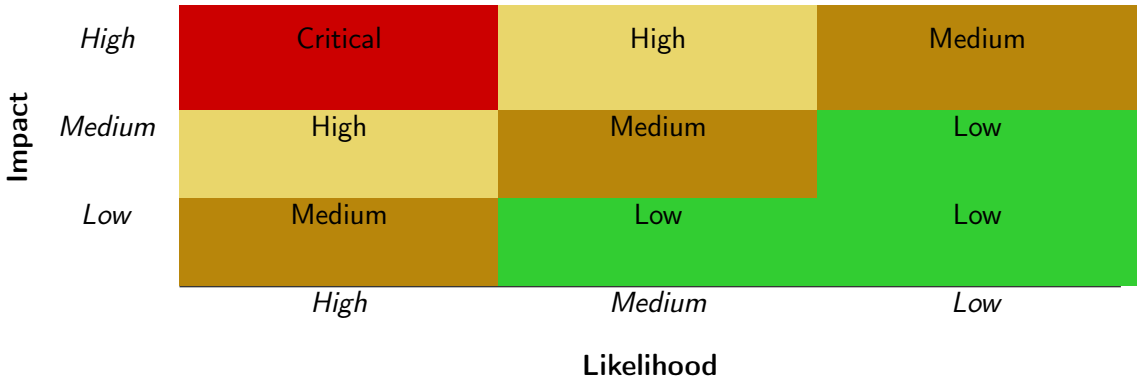
In the following, we show the audited contract code deployed at the BSC chain with the following address:

- https://bscscan.com/address/0x9e6b3e35c8f563b45d864f9ff697a144ad28a371#code

1.2 About PeckShield

PeckShield Inc. [12] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification



1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [11]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [10], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the `DOGOToken` implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	■■
Low	2	■■
Informational	1	■
Undetermined	0	
Total	5	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key DOGOToken Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Potential Sandwich/MEV Attack In DOGOToken	Time and State	Confirmed
PVE-002	Informational	Improved Validation Of Function Arguments	Coding Practices	Confirmed
PVE-003	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed
PVE-004	Low	Potential Lockup Of ETH Leftover In swapAndLiquify()	Business Logics	Confirmed
PVE-005	Low	Inconsistent Logic Between State Initialization And Their Updates	Business Logics	Confirmed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Potential Sandwich/MEV Attack In DOGOToken

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `DOGOToken`
- Category: Time and State [9]
- CWE subcategory: CWE-682 [3]

Description

While examining the `DOGOToken` contract, we notice there are several functions that can be improved with slippage control. In the following, we take the `swapTokensForEth()` routine as an example. To elaborate, we show below the related code snippet of the `DOGOToken` contract. According to the design, the `swapTokensForEth()` function is used to swap `DOGOToken` to `WETH`. In the function, the `swapExactTokensForETHSupportingFeeOnTransferTokens()` function of `UniswapV2` is called (line 1790) to swap the exact `DOGOToken` to `WETH`. However, we observe the second input `amountOutMin` parameter is assigned to `0`, which means this transaction does not specify any restriction on possible slippage and is therefore vulnerable to possible front-running attacks.

```
1779    function swapTokensForEth(uint256 tokenAmount) private {


1782        // generate the uniswap pair path of token -> weth
1783        address[] memory path = new address[](2);
1784        path[0] = address(this);
1785        path[1] = uniswapV2Router.WETH();

1787        _approve(address(this), address(uniswapV2Router), tokenAmount);

1789        // make the swap
1790        uniswapV2Router.swapExactTokensForETHSupportingFeeOnTransferTokens(
1791            tokenAmount,
1792            0, // accept any amount of ETH
1793            path,
```

```
1794                address(this),
1795                block.timestamp
1796            );

1798        }
```

Listing 3.1: DOGOToken::swapTokensForEth()

Note the swapTokensForDoge() and addLiquidity() routines can be similarly improved.

Recommendation Improve the above-mentioned functions by adding necessary slippage control.

Status The issue has been confirmed by the team.

3.2 Improved Validation Of Function Arguments

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: DOGOToken
- Category: Coding Practices [7]
- CWE subcategory: CWE-628 [2]

Description

In the DOGOToken contract, we observe there are three functions that are designed to update protocol-wide fee parameters, i.e., DOGERewardsFee, liquidityFee and marketingFee. In the following, we take the setDOGERewardsFee() routine as an example and show below the related code snippet.

This setDOGERewardsFee() function is designed to update the DOGERewardsFee and the precision of the fee is 100. In the setDOGERewardsFee() function, we notice the input value is directly stored into the DOGERewardsFee storage variable (line 1539) and the totalFees storage variable is updated (line 1540) subsequently without any validation. This is reasonable under the assumption that the input value parameter is always correctly provided and the totalFees storage variable is less than 100. However, in the unlikely situation, if the value is improperly provided to result in the totalFees larger than 100, the transactions related with the DOGOToken transfer will be reverted.

```
1538    function setDOGERewardsFee(uint256 value) external onlyOwner{
1539        DOGERewardsFee = value;
1540        totalFees = DOGERewardsFee.add(liquidityFee).add(marketingFee);
1541    }
```

Listing 3.2: DOGOToken::setDOGERewardsFee()

Note the setLiquiditFee() and setMarketingFee() routines can be similarly improved.

Recommendation Add necessary validation for above-mentioned routines to ensure the `totalFees` is less than `100`.

Status The issue has been confirmed. And the team plans to exercise extra caution in properly configuring them.

3.3 Trust Issue Of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `DOGOToken`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [1]

Description

In the `DOGOToken` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configuring various system parameters). In the following, we show the representative functions potentially affected by the privilege of the `owner` account.

```
1490    function updateDividendTracker(address newAddress) public onlyOwner {
1491        require(newAddress != address(dividendTracker), "DOGO: The dividend tracker
                already has that address");
1492
1493        DOGODividendTracker newDividendTracker = DOGODividendTracker(payable(newAddress)
                );
1494
1495        require(newDividendTracker.owner() == address(this), "DOGO: The new dividend
                tracker must be owned by the DOGO token contract");
1496
1497        newDividendTracker.excludeFromDividends(address(newDividendTracker));
1498        newDividendTracker.excludeFromDividends(address(this));
1499        newDividendTracker.excludeFromDividends(owner());
1500        newDividendTracker.excludeFromDividends(address(uniswapV2Router));
1501
1502        emit UpdateDividendTracker(newAddress, address(dividendTracker));
1503
1504        dividendTracker = newDividendTracker;
1505    }
1506
1507    ...
1508
1509    function blacklistAddress(address account, bool value) external onlyOwner{
1510        _isBlacklisted[account] = value;
1511    }
```

Listing 3.3: `DOGOToken::updateDividendTracker()&&blacklistAddress()`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged `owner` account is not governed by a `DAO`-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the `DOGOToken` design.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed by the team.

3.4 Potential Lockup Of ETH Leftover In swapAndLiquify()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `DOGOToken`
- Category: Business Logics [8]
- CWE subcategory: CWE-754 [4]

Description

According to the `DOGOToken` design, for each transfer transaction, the certain percentage of the transfer amount specified by the `liquidityFee` storage variable will be used to add liquidity for the `UniswapV2 DOGOToken + WETH pair`. While examining the process of adding liquidity, we notice the leftover ETH may be locked in the `DOGOToken` contract.

To elaborate, we show below the related code snippet of the `DOGOToken` contract. The `swapAndLiquify()` function is designed to add liquidity for the `UniswapV2 DOGOToken + WETH pair`. In the function, it comes to our attention with the following action sequences: The `swapTokensForEth(half)` is firstly called (line 1767) to swap half the number of the `DOGOToken` (calculated from the input `tokens` parameter) to `WETH` and then the `addLiquidity(otherHalf, newBalance)` is called (line 1773) to add the remaining half of the `DOGOToken` and the exchanged `WETH` to the `UniswapV2 DOGOToken + WETH pair` to provide liquidity. However, the logic ignores the fact that the calling of `swapTokensForEth(half)` (line 1767) will drive up the price of the `WETH` in the pair, which will lead to the result where certain `WETH` is left after the calling of `addLiquidity(otherHalf, newBalance)` (line 1773).

```
1755    function swapAndLiquify(uint256 tokens) private {
1756        // split the contract balance into halves
1757        uint256 half = tokens.div(2);
1758        uint256 otherHalf = tokens.sub(half);
1759
```

```
1760        // capture the contract's current ETH balance.
1761        // this is so that we can capture exactly the amount of ETH that the
1762        // swap creates, and not make the liquidity event include any ETH that
1763        // has been manually sent to the contract
1764        uint256 initialBalance = address(this).balance;
1765
1766        // swap tokens for ETH
1767        swapTokensForEth(half); // <- this breaks the ETH -> HATE swap when swap+liquify
                is triggered
1768
1769        // how much ETH did we just swap into?
1770        uint256 newBalance = address(this).balance.sub(initialBalance);
1771
1772        // add liquidity to uniswap
1773        addLiquidity(otherHalf, newBalance);
1774
1775        emit SwapAndLiquify(half, newBalance, otherHalf);
1776    }
```

Listing 3.4: `DOGOToken::swapAndLiquify()`

Recommendation Apply the below optimal mechanism to calculate the swap amount rather than using half the number of the `DOGOToken` directly.

```
1755    /// @param amtA amount of token A desired to deposit
1756    /// @param amtB amount of token B desired to deposit
1757    /// @param resA amount of token A in reserve
1758    /// @param resB amount of token B in reserve
1759    // e - b / a * 2
1760    // Math.sqrt((b * b) + d) - b / 9970 * 2
1761    // (19970 * resA) * (19970 * resA) + (a*c*4) / 19950
1762
1763    // e-b / 9970
1764    function _optimalDepositA(
1765        uint256 amtA,
1766        uint256 amtB,
1767        uint256 resA,
1768        uint256 resB
1769    ) private pure returns (uint256) {
1770        require(amtA * resB >= amtB * resA, "Reversed");
1771
1772        uint256 a = 997;    // change fee here
1773        uint256 b = 1997 * resA;    // change fee here
1774        uint256 _c = (amtA * resB) - (amtB * resA);
1775        uint256 c = ((_c * 1000) / (amtB + resB)) * resA;
1776
1777        uint256 d = a * c * 4;
1778        uint256 e = Babylonian.sqrt((b * b) + d);
1779
1780        uint256 numerator = e - b;
1781        uint256 denominator = a * 2;
1782
```

```
1783        return numerator / denominator;
1784    }
```

Listing 3.5: DOGOToken::swapAndLiquify()

Status The issue has been confirmed.

3.5 Inconsistent Logic Between State Initialization And Their Updates

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: DOGOToken
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

In the DOGOToken protocol, those specific DOGOToken holders each with the balance larger than the system-specified threshold minimumTokenBalanceForDividends will receive the DOGE token as the dividends. The dividendTracker storage variable is used to locate the actual DOGODividendTracker contract that is designed to track the dividends for those specific DOGOToken holders.

To elaborate, we show below the related code snippet of the DOGOToken contract. While examining the logic of the updateDividendTracker() function, we notice the deadWallet which is a specific address usually used to burn tokens is not excluded from the dividends. However, we observe it is excluded from the dividends (line 1458) in the constructor() function. We suggest to keep the implementation consistent between the constructor() function and the updateDividendTracker() function.

```
1446    constructor() public ERC20("DogemonGo", "DOGO") {
1447
1448      dividendTracker = new DOGODividendTracker();
1449
1450
1451      IUniswapV2Router02 _uniswapV2Router = IUniswapV2Router02(0
              x10ED43C718714eb63d5aA57B78B54704E256024E);
1452        uniswapV2Router = _uniswapV2Router;
1453
1454        // exclude from receiving dividends
1455        dividendTracker.excludeFromDividends(address(dividendTracker));
1456        dividendTracker.excludeFromDividends(address(this));
1457        dividendTracker.excludeFromDividends(owner());
1458        dividendTracker.excludeFromDividends(deadWallet);
1459        dividendTracker.excludeFromDividends(address(_uniswapV2Router));
1460
```

```
1461        // exclude from paying fees or having max transaction amount
1462        excludeFromFees(owner(), true);
1463        excludeFromFees(_marketingWalletAddress0, true);
1464        excludeFromFees(_marketingWalletAddress1, true);
1465        excludeFromFees(_marketingWalletAddress2, true);
1466        excludeFromFees(_marketingWalletAddress3, true);
1467        excludeFromFees(address(this), true);
1468
1469        /*
1470            _mint is an internal function in ERC20.sol that is only called here,
1471            and CANNOT be called ever again
1472        */
1473        _mint(owner(), 100000000000 * (10**18));
1474    }
```

Listing 3.6: DOGOToken::constructor()

```
1490    function updateDividendTracker(address newAddress) public onlyOwner {
1491        require(newAddress != address(dividendTracker), "DOGO: The dividend tracker
                already has that address");
1492
1493        DOGODividendTracker newDividendTracker = DOGODividendTracker(payable(newAddress)
                );
1494
1495        require(newDividendTracker.owner() == address(this), "DOGO: The new dividend
                tracker must be owned by the DOGO token contract");
1496
1497        newDividendTracker.excludeFromDividends(address(newDividendTracker));
1498        newDividendTracker.excludeFromDividends(address(this));
1499        newDividendTracker.excludeFromDividends(owner());
1500        newDividendTracker.excludeFromDividends(address(uniswapV2Router));
1501
1502        emit UpdateDividendTracker(newAddress, address(dividendTracker));
1503
1504        dividendTracker = newDividendTracker;
1505    }
```

Listing 3.7: DOGOToken::updateDividendTracker()

Moreover, according to the DOGOToken design, a certain percentage of the transfer amount will be treated as the transaction fee, including DOGERewardsFee, liquidityFee and marketingFee. The privileged owner account can exclude some specific addresses from the transaction fees. We notice the four marketing wallets specified by _marketingWalletAddress0, _marketingWalletAddress1, _marketingWalletAddress2 and _marketingWalletAddress3 are excluded from being charged for the transaction fee in the constructor() function. However, the new marketing wallets in the setMarketingWallet() function are not.

```
1531    function setMarketingWallet(address payable wallet0, address payable wallet1,
            address payable wallet2, address payable wallet3) external onlyOwner{
1532        _marketingWalletAddress0 = wallet0;
```

```
1533        _marketingWalletAddress1 = wallet1;
1534        _marketingWalletAddress2 = wallet2;
1535        _marketingWalletAddress3 = wallet3;
1536    }
```

Listing 3.8: `DOGOToken::setMarketingWallet()`

Recommendation We suggest to keep the implementation consistent between the `constructor()` function and the `updateDividendTracker()` function.

Status The issue has been confirmed by the team. Considering the code is alive on the mainnet, the team intends to use the public `excludeFromFees()` and `excludeFromDividends()` functions to reach the purpose.

4 | Conclusion

In this audit, we have analyzed the DOGOToken design and implementation. The DOGOToken is widely used in the DogemonGo game, which is a completely free crypto game that makes use of Augmented Reality (AR) to deliver an interactive experience. The DogemonGo game is an unique combination of game, meme and crypto-culture, which enriches the DeFi ecosystem and also presents a unique contribution to current DeFi ecosystem. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.html.

[3] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[4] MITRE. CWE-754: Improper Check for Unusual or Exceptional Conditions. https://cwe.mitre.org/data/definitions/754.html.

[5] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[6] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[7] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[8] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[9] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[10] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[11] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[12] PeckShield. PeckShield Inc. https://www.peckshield.com.