Smart Contract Audit Report

April, 2023

DragonBallDAO

DEFIMOON PROJECT

Audit and Development

CONTACTS

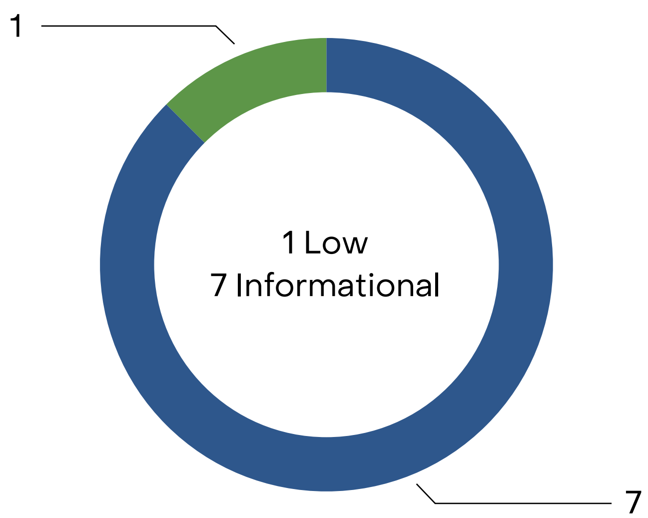
defimoon.org
audit@defimoon.org
defimoon_org
defimoonorg
defimoon
defimoonorg

6 May 2023

This audit report was prepared by DefiMoon for DragonBallDAO.

Audit information

Description	ERC20 token with advanced trading functionality and liquidity management.
Audited files	DragonBallDAO.sol
Timeline	6 May 2023
Audited by	Ilya Vaganov
Approved by	Artur Makhnach, Kirill Minyaev
Languages	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Manual Review
Source code	https://etherscan.io/address/0x2b9e59988b77a77cfae04415060f00c044342c15#code
Chain	Ethereum
Status	Passed



	High Risk	A fatal vulnerability that can cause the loss of all Tokens / Funds.
	Medium Risk	A vulnerability that can cause the loss of some Tokens / Funds.
	Low Risk	A vulnerability which can cause the loss of protocol functionality.
	Informational	Non-security issues such as functionality, style, and convention.

Disclaimer

This audit is not financial, investment, or any other kind of advice and could be used for informational purposes only. This report is not a substitute for doing your own research and due diligence should always be paid in full to any project. Defimoon is not responsible or liable for any loss, damage, or otherwise caused by reliance on this report for any purpose. Defimoon has based this audit report solely on the information provided by the audited party and on facts that existed before or during the audit being conducted. Defimoon is not responsible for any outcome, including changes done to the contract/contracts after the audit was published. This audit is fully objective and only discerns what the contract is saying without adding any opinion to it. Defimoon has no connection to the project other than the conduction of this audit and has no obligations other than to publish an objective report. Defimoon will always publish its findings regardless of the outcome of the findings. The audit only covers the subject areas detailed in this report and unless specifically stated, nothing else has been audited. Defimoon assumes that the provided information and materials were not altered, suppressed, or misleading. This report is published by Defimoon, and Defimoon has sole ownership of this report. Use of this report for any reason other than for informational purposes on the subjects reviewed in this report including the use of any part of this report is prohibited without the express written consent of Defimoon. In instances where an auditor or team member has a personal connection with the audited project, that auditor or team member will be excluded from viewing or impacting any internal communication regarding the specific audit.

Audit Information

Defimoon utilizes both manual and automated auditing approach to cover the most ground possible. We begin with generic static analysis automated tools to quickly assess the overall state of the contract. We then move to a comprehensive manual code analysis, which enables us to find security flaws that automated tools would miss. Finally, we conduct an extensive unit testing to make sure contract behaves as expected under stress conditions.

In our decision making process we rely on finding located via the manual code inspection and testing. If an automated tool raises a possible vulnerability, we always investigate it further manually to make a final verdict. All our tests are run in a special test environment which matches the "real world" situations and we utilize exact copies of the published or provided contracts.

While conducting the audit, the Defimoon security team uses best practices to ensure that the reviewed contracts are thoroughly examined against all angles of attack. This is done by evaluating the codebase and whether it gives rise to significant risks. During the audit, Defimoon assesses the risks and assigns a risk level to each section together with an explanatory comment.

Audit overview

Major issues were not found.

To comply with the best practices for developing smart contracts, you can refuse to use SafeMath, supplement the NatSpec code with comments and specify a license for the contract, for example, MIT.

Summary of findings

ID	Description	Severity
DFM-1	No checks when changing fees	Low Risk
DFM-2	Pointless use of SafeMath	Informational
DFM-3	Description does not match code	Informational
DFM-4	Percentages do not change values dynamically	Informational
DFM-5	No additional condition	Informational
DFM-6	Optimization of calculations	Informational
DFM-7	Gas optimization	Informational
DFM-8	Note	Informational

Application security checklist

Compiler errors	Passed
Possible delays in data delivery	Passed
Timestamp dependence	Passed
Integer Overflow and Underflow	Passed
Race Conditions and Reentrancy	Passed
DoS with Revert	Passed
DoS with block gas limit	Passed
Methods execution permissions	Passed
Private user data leaks	Passed
Malicious Events Log	Passed
Scoping and Declarations	Passed
Uninitialized storage pointers	Passed
Arithmetic accuracy	Passed
Design Logic	Passed
Cross-function race conditions	Passed

Detailed Audit Information

Contract Programming

Solidity version not specified	Passed
Solidity version too old	Passed
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	Passed
Fallback function misuse	Passed
Race condition	Passed
Logical vulnerability	Passed
Other programming issues	Passed

Code Specification

Visibility not explicitly declared	Passed
Variable 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

Findings

DFM-1 «No checks when changing fees»

Severity: Low Risk

Description: Fees limits are not set in the setFees_base1000 function. As a result, fees may be set incorrectly in such a way that their totalFee amount will exceed 100% due to which errors may occur or some fees may turn out to be unreasonably high.

Recommendation: The best practice is to set limits for each of the fees and for the totalFee in such a way that they do not exceed certain values, similar to how it is implemented in the update_fees function for multipliers.

DFM-2 «Pointless use of SafeMath»

Severity: Informational

Description: Since version 0.8.0, the definition of overflow and underflow of variables is built into the Solidity compiler and the use of the SafeMath library does not make sense, but only takes up the contract bytecode. You are using version 0.8.19.

Recommendation: You can replace using the SafeMath library with regular arithmetic operations.

DFM-3 «Description does not match code»

Severity: Informational

Description: The clearStuckToken function in require has a limit of 500 days, even though the error description says 1 year.

```
require(block.timestamp > launchedAt + 500 days,"Locked for 1 year");
```

Recommendation: Please make sure that the code matches the intended logic and fix the error in the condition or in the error description.

DFM-4 «Percentages do not change values dynamically»

Severity: Informational

Description: The setMaxWalletPercent_base10000 and setMaxTxPercent_base10000 functions take a percentage value as an argument and update the storage variables relative to the current _totalSupply. But _totalSupply can change over time when burning tokens.

Recommendation: Please make sure that the code matches the intended logic. If the values of _maxWalletToken and _maxTxAmount still need to change relative to _totalSupply dynamically, then fix it.

DFM-5 «No additional condition»

Severity: Informational

Description: In the swapBack function, the value of amountBNBLiquidity can theoretically be equal to zero, which can cause an error when adding liquidity.

Recommendation: The best practice would be to add one more condition necessary to add liquidity.

```
if (amountToLiquify > 0 && amountBNBLiquidity > 0)
```

DFM-6 «Optimization of calculations»

Severity: Informational

Description: In the swapBack function, the amountBNBLiquidity, amountBNBMarketing and amountBNBLiquidity values can be calculated in such a way that their sum is not equal to amountBNB. This is because Solidity uses whole numbers and rounds down.

Recommendation: The best practice would be to get the remainder for one or more values, like this:

```
uint256 amountBNBMarketing = (amountBNB * marketingFee) / totalETHFee;
uint256 amountBNBbuyback = (amountBNB * treasuryFee) / totalETHFee;
uint256 amountBNBLiquidity = amountBNB - amountBNBMarketing - amountBNBbuyback;
```

DFM-7 «Gas optimization»

Severity: Informational

Description: The manage_FeeExempt, manage_TxLimitExempt, and manage_WalletLimitExempt functions use loops with a potentially large number of iterations, which can result in high gas usage.

Recommendation: The best gas optimization solution would be to bring the functions to this format:

```
function manage_FeeExempt(address[] calldata addresses, bool status) external authorized {
  uint256 l = addresses.length;
  require(l < 501, "GAS Error: max limit is 500 addresses");
  for (uint256 i; i < l; ) {
    isFeeExempt[addresses[i]] = status;
    emit Wallet_feeExempt(addresses[i], status);
    unchecked { ++i; }
  }
}
```

DFM-8 «Note»

Severity: Informational

Description: The _transferFrom function contains logic that is not entirely clear to us:

```
if (!authorizations[sender] && !isWalletLimitExempt[sender] && !isWalletLimitExempt[recipient]
&& recipient != pair) {
   require((balanceOf[recipient] + amount) <= _maxWalletToken, "max wallet limit reached");
}
```

The require checks against the recipient, but the condition for this check also includes checks against the sender, for example: !authorizations[sender] and !isWalletLimitExempt[sender]. Thus, if at least one of the conditions is false, then the require will not be executed. For example, if the sender is exempt from the limit, then this check will not be performed for the recipient, although it is he who receives the tokens.

Due to the lack of detailed comments in the code or protocol whitepaper, we cannot know for sure whether this logic is correct or not.

Recommendation: Please make sure that the code matches the intended logic and fix the error in the condition, if it exists.

Automated Analyses

Slither

Slither's automatic analysis not found vulnerabilities, or these false positives results .

Methodology

Manual Code Review

We prefer to work with a transparent process and make our reviews a collaborative effort. The goal of our security audits is 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.

Vulnerability Analysis

Our audit techniques include 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, review open issue tickets, and 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 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 to make a final decision.

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.

Appendix A — Finding Statuses

Resolved	Contracts were modified to permanently resolve the finding
Mitigated	The finding was resolved by other methods such as revoking contract ownership or updating the code to minimize the effect of the finding
Acknowledged	Project team is made aware of the finding
Open	The finding was not addressed