DEFIMOON

be secure

Smart Contract Audit Report

November, 2022

BabyCake

DEFIMOON PROJECT

Audit and
Development

CONTACTS

https://defimoon.org
audit@defimoon.org
defimoon_org
defimoonorg
defimoon
defimoonorg

DEFIMOON

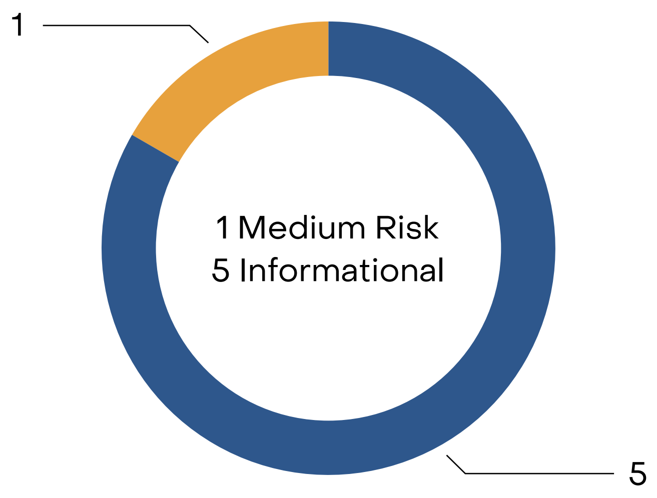
be secure

19 November 2022

This audit report was prepared by DefiMoon for BabyCake

Audit information

Description	ERC20 (BEP20) Token contract
Audited files	Token smart contract
Timeline	18 - 19 November 2022
Audited by	Ilya Vaganov
Approved by	Artur Makhnach, Kirill Minyaev
Languages	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Manual Review
Source code	https://bscscan.com/address/0x194c44031C1c8746E742d5D13FeC9CC35B0F6B2b#code
Chain	Binance Smart Chain (BSC)
Status	Not 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

No major issues were found, however issues raised may lead to the loss of the contract.

There are moments that can be optimized for gas and make the code more readable and understandable for third parties.

The currently outdated SafeMath library is used.

Most of the functionality is the functionality of the ERC20 (BEP20) token standard and can be implemented using solutions from OpenZeppelin. Remember, using ready-made solutions that have passed many tests and are used everywhere has a positive effect on the work of your code.

Some best practices could also be implemented:

1. Look towards ready-made solutions, if possible.
2. Write code according to the capabilities of your compiler version.
3. For now, it's best to stick with version 0.8.13 and not use newer versions until they've been more thoroughly tested by users and developers.

Check list

Description	Status
No mint function found, owner cannot mint tokens after initial deploy	✅
Owner can't set max tx amount	✅
Owner can't set fees over 25%	✅
Owner can't pause trading	✅
Owner can't blacklist wallets	✅

Summary of findings

According to the standard audit evaluation, Solidity's validated smart contracts are fairly secure, but there are still vulnerabilities present, gas consumption can be slightly optimized. It is recommended to fully secure the codebase before production.

ID	**Description**	**Severity**
DFM-1	Possible to lose ownership	Medium Risk
DFM-2	Pointless use of SafeMath	Informational
DFM-3	Circulating Supply may not be calculated as expected	Informational
DFM-4	Useless variables	Informational
DFM-5	Preferred way to send ETH	Informational
DFM-6	Gas and code optimizations	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	Not 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	Not 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 «Possible to lose ownership»

Severity: Medium Risk

Description:
The implementation of the Ownable contract includes the `waiveOwnership()` function, with which you can lose the owner of a smart contract if you are not careful. Your smart contract does not require the owner to opt out, because there is functionality intended only for the owner.

You can also lose ownership by calling the `transferOwnership()` function by accidentally transferring ownership to the wrong address.

Recommendation:
The simplest solution would be to override the `waiveOwnership()` function like this:

```
function waiveOwnership() public override onlyOwner {
    revert("Cant renounce ownership");
}
```

If necessary, `transferOwnership()` can do the same, or you can use the pending owner concept to avoid transferring ownership to the wrong address.

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 0.8.15.

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

DFM-3 «Circulating Supply may not be calculated as expected»

Severity: Informational

Description: As we understand it, the Token::`getCirculatingSupply()` function must return a valid supply of tokens that were not burned by sending them to the `dEaD` address. This value may not be calculated as it actually is, since there is more than one address for burning tokens.

Recommendation: In general, there is nothing to worry about, this is a common practice, but it is worth keeping this in mind. Do not use the result of executing this function in calculations that require high precision.

DFM-4 «Useless variables»

Severity: Informational

Description: The variables Token::`swapAndLiquifyEnabled` and Token::`swapAndLiquifyByLimitOnly` are used in if-else constructs, but they always take the only value that is set during initialization and cannot be changed later. Thus, the use of these variables in if-else constructs and in the contract as a whole ceases to be necessary, because their value is known in advance.

Recommendation: These variables can be removed and the code rewritten to reflect these changes. Like this:

```
function _transfer(address sender, address recipient, uint256 amount) private returns
(bool) {
…
    if (overMinimumTokenBalance && !inSwapAndLiquify && !isMarketPair[sender] &&
recipient!=owner()) {
        swapAndLiquify(contractTokenBalance);
    }
…
```

DFM-5 «Preferred way to send ETH»

Severity: Informational

Description: To send ether, it is better to use the `call()` method, which, among other things, allows you to return your own readable error.

Recommendation: You can see an example in the TransferHelper library.

DFM-6 «Gas and code optimizations»

Severity: Informational

Description: We want to offer you several gas-safe moments to optimize gas consumption when deploying a smart contract and interacting with it.

Recommendation:

1) You can use the "`constant`" keyword for variables that are initialized on creation and do not change in the future (for example Token::`_name`, Token::`_symbol`, Token::`_decimals`, Token::`marketingWallet`, Token::`deadAddress` ...).

2) You can use the "`immutable`" keyword for variables that are initialized in the constructor and do not change in the future (for example Token::`uniswapV2Router`, Token::`uniswapPair`).

   Also, Token::`uniswapV2Router` can be initialized when creating a variable and declared as `constant`.

3) You are using two variables that store the same value. Token::`_marketingFee` and Token::`_totalTax` which is always equal to `_marketingFee`. One of the variables can be removed.

   In addition, the variable Token::`_totalTax` is assigned the same value twice: at initialization and in the constructor.

   In addition, null values (for any data type) are specified for default variables. For example, `0` for `uint`, `false` for `bool`, and so on, so they can be omitted during creation.

4) Extra local variable Token::`_uniswapV2Router` in the constructor. You can immediately use Token::`uniswapV2Router`.

5) You are using repetitive functionality for functions declared in a token interface. For example, Token::`totalSupply()` and Token::`_totalSupply`, Token::`name()` and Token::`_name` and so on). You can use the `name` variable instead of Token::`_name` with the "`public`" scope, then a getter will be automatically created for it when the smart contract is compiled. Similarly for other similar variables. Like this:

```
…
uint256 public override totalSupply = 100_000_000 * 10 ** _decimals;
uint256 public minimumTokensBeforeSwap = 1000 * 10 ** _decimals;
…
```

6) Unused events `SwapETHForTokens` and `SwapTokensForETH`, `SwapAndLiquifyEnabledUpdated`, `SwapAndLiquify`.

7) The useless `!inSwapAndLiquify` check in the if-else construct in the Token::`_transfer()` function. The `if` already checks for `inSwapAndLiquify`, so if the execution goes to `else` it already means that `!inSwapAndLiquify = true`. Given the **DF-4** will look like this:

```
function _transfer(address sender, address recipient, uint256 amount) private returns
(bool) {
…
    if (overMinimumTokenBalance && !isMarketPair[sender] && recipient!=owner()) {
        swapAndLiquify(contractTokenBalance);
    }
…
```

8) You can use the already existing Token::`_basicTransfer()` function in the else part of the Token::`_transfer()` function. Like this:

```
    else {
    …
        uint256 finalAmount = (isExcludedFromFee[sender] || isExcludedFromFee[recipient])
? amount : takeFee(sender, recipient, amount);
        return _basicTransfer(sender, recipient, finalAmount);
    }
```

9) Useless approve in the constructor:

```
_allowances[address(this)][address(uniswapV2Router)] = _totalSupply;
```

because in the Token::`swapTokensForBNB()` function it always does approve before swapping tokens.

Automated Analyses

Slither

Slither's automatic analysis found no additional vulnerabilities, or these results were either related to code from dependencies or false positives.

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

Closed	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