



Request your audit at [coinsult.net](https://coinsult.net)

Advanced Manual

# Smart Contract Audit

November 10, 2022



Audit requested by

**Check Me Finance**

0xd1b5c3031c1b107fa7db2dfba14eba83a25d0202

# Table of Contents

Table of Contents	2
Audit Summary	4
Audit Scope	5
Source Code	5
Audit Method	5
Automated Vulnerability Check	5
Manual Code Review	5
Used Tools	5
Static Analysis	6
Audit Results	8
Risk Classification	8
Manual Code Review	8
Presence of unused variables	9
Contract does not use a ReEntrancyGuard	10
Floating Pragma	11
Initial Supply	12
Reliance on third-parties	13
Too many digits	14
Unchecked Call Return Value	15
Missing events arithmetic	16
State variable visibility is not set	17
Centralization: Trading fee limitation	18
Centralization: Pause trading	19
Centralization: Max transaction amount	20
Centralization: Excluding from fees	21
Centralization: Mint after deployment	22
Centralization: Ability to blacklist	23
Centralization Privileges	24
Notes	25

Notes by CMF	25
Notes by Coinsult	25
Constructor Snapshot	26
Disclaimer	27

# Audit Summary

Project Name	Green Baby Floki
Website	<a href="https://checkme.finance/">https://checkme.finance/</a>
Blockchain	Binance Smart Chain
Smart Contract Language	Solidity
Contract Address	0xd1b5c3031c1b107fa7db2dfba14eba83a25d0202
Audit Method	Static Analysis, Manual Review
Start date of audit	November 10, 2022

This audit report has been prepared by Coinsult's experts at the request of the client. In this audit, the results of the static analysis and the manual code review will be presented. The purpose of the audit is to see if the functions work as intended, and to identify potential security issues within the smart contract.

The information in this report should be used to understand the risks associated with the smart contract. This report can be used as a guide for the development team on how the contract could possibly be improved by remediating the issues that were identified.

# Audit Scope

## Source Code

Coinsult was commissioned by Check Me Finance to perform an audit based on the following code:

<https://bscscan.com/token/0xd1b5c3031c1b107fa7db2dfba14eba83a25d0202#code>

Note that we only audited the code available to us on this URL at the time of the audit. If the URL is not from any block explorer (main net), it may be subject to change. Always check the contract address on this audit report and compare it to the token you are doing research for.

## Audit Method

Coinsult's manual smart contract audit is an extensive methodical examination and analysis of the smart contract's code that is used to interact with the blockchain. This process is conducted to discover errors, issues and security vulnerabilities in the code in order to suggest improvements and ways to fix them.

## Automated Vulnerability Check

Coinsult uses software that checks for common vulnerability issues within smart contracts. We use automated tools that scan the contract for security vulnerabilities such as integer-overflow, integer-underflow, out-of-gas-situations, unchecked transfers, etc.

## Manual Code Review

Coinsult's manual code review involves a human looking at source code, line by line, to find vulnerabilities. Manual code review helps to clarify the context of coding decisions. Automated tools are faster but they cannot take the developer's intentions and general business logic into consideration.

## Used Tools

- ✓ Slither: Solidity static analysis framework
- ✓ Remix: IDE Developer Tool
- ✓ CWE: Common Weakness Enumeration
- ✓ SWC: Smart Contract Weakness Classification and Test Cases
- ✓ DEX: Testnet Blockchains

# Static Analysis

The Smart Contract Weakness Classification Registry (SWC Registry) is an implementation of the weakness classification scheme proposed in EIP-1470. It is loosely aligned to the terminologies and structure used in the Common Weakness Enumeration (CWE) while overlaying a wide range of weakness variants that are specific to smart contracts.





ID	Description	Status
SWC-100	Function Default Visibility	Passed
SWC-101	Integer Overflow and Underflow	Passed
SWC-102	Outdated Compiler Version	Passed
SWC-103	Floating Pragma	Failed
SWC-104	Unchecked Call Return Value	Passed
SWC-105	Unprotected Ether Withdrawal	Passed
SWC-106	Unprotected SELFDESTRUCT Instruction	Passed
SWC-107	Reentrancy	Passed
SWC-108	State Variable Default Visibility	Failed
SWC-109	Uninitialized Storage Pointer	Passed
SWC-110	Assert Violation	Passed
SWC-111	Use of Deprecated Solidity Functions	Passed
SWC-112	Delegatecall to Untrusted Callee	Passed
SWC-113	DoS with Failed Call	Passed
SWC-114	Transaction Order Dependence	Passed
SWC-115	Authorization through tx.origin	Passed
SWC-116	Block values as a proxy for time	Passed

SWC-117	Signature Malleability	Passed
SWC-118	Incorrect Constructor Name	Passed
SWC-119	Shadowing State Variables	Passed
SWC-120	Weak Sources of Randomness from Chain Attributes	Passed
SWC-121	Missing Protection against Signature Replay Attacks	Passed
SWC-122	Lack of Proper Signature Verification	Passed
SWC-123	Requirement Violation	Passed
SWC-124	Write to Arbitrary Storage Location	Passed
SWC-125	Incorrect Inheritance Order	Passed
SWC-126	Insufficient Gas Griefing	Passed
SWC-127	Arbitrary Jump with Function Type Variable	Passed
SWC-128	DoS With Block Gas Limit	Passed
SWC-129	Does not compromise the functionality of the contract in any way	Passed
SWC-130	Does not compromise the functionality of the contract in any way	Passed
SWC-131	Does not compromise the functionality of the contract in any way	Passed
SWC-132	Does not compromise the functionality of the contract in any way	Passed
SWC-133	Does not compromise the functionality of the contract in any way	Passed
SWC-134	Does not compromise the functionality of the contract in any way	Passed
SWC-135	Does not compromise the functionality of the contract in any way	Passed
SWC-136	Will definitely cause problems, this needs to be adjusted	Passed

# Audit Results





## Risk Classification

Coinsult uses certain vulnerability levels, these indicate how bad a certain issue is. The higher the risk, the more strictly it is recommended to correct the error before using the contract.

Vulnerability Level	Description
 Informational	Does not compromise the functionality of the contract in any way
 Low-Risk	Won't cause any problems, but can be adjusted for improvement
 Medium-Risk	Will likely cause problems and it is recommended to adjust
 High-Risk	Will definitely cause problems, this needs to be adjusted

## Manual Code Review

In this audit report we will highlight the following issues:

Vulnerability Level	Total	Pending	Acknowledged	Resolved
 Informational	1	1	0	0
 Low-Risk	8	8	0	0
 Medium-Risk	0	0	0	0
 High-Risk	0	0	0	0



Error Code	Description	Severity
SWC-131	CWE-1164: Irrelevant Code	<span>●</span> Informational

## Presence of unused variables

Unused variables are allowed in Solidity and they do not pose a direct security issue. It is best practice though to avoid them as they can: cause an increase in computations (and unnecessary gas consumption), indicate bugs or malformed data structures and they are generally a sign of poor code quality or cause code noise and decrease readability of the code

```
uint256 burnFee = 2;
```

## Recommendation

Remove all unused variables from the code base.

Error Code	Description	Severity
SWC-107	CWE-841: Improper Enforcement of Behavioral Workflow	● Low

### Contract does not use a ReEntrancyGuard

One of the major dangers of calling external contracts is that they can take over the control flow. In the reentrancy attack (a.k.a. recursive call attack), a malicious contract calls back into the calling contract before the first invocation of the function is finished. This may cause the different invocations of the function to interact in undesirable ways.

### Recommendation

The best practices to avoid Reentrancy weaknesses are: Make sure all internal state changes are performed before the call is executed. This is known as the Checks-Effects-Interactions pattern, or use a reentrancy lock (ie. OpenZeppelin's ReentrancyGuard).

Error Code	Description	Severity
SWC-103	CWE-664: Improper Control of a Resource Through its Lifetime	● Low

## Floating Pragma

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

```
pragma solidity ^0.8.17;
```

## Recommendation

Lock the pragma version and also consider known bugs (<https://github.com/ethereum/solidity/releases>) for the compiler version that is chosen.

Pragma statements can be allowed to float when a contract is intended for consumption by other developers, as in the case with contracts in a library or EthPM package. Otherwise, the developer would need to manually update the pragma in order to compile locally.

Error Code	Description	Severity
CNS-016	Initial supply	 Low

## Initial Supply

When the contract is deployed, the contract deployer receives all of the initially created assets. Since the deployer and/or contract owner can distribute tokens without consulting the community, this could be a problem.

## Recommendation

Private keys belonging to the employer and/or contract owner should be stored properly. The initial asset allocation procedure should involve consultation with the community.

Error Code	Description	Severity
CNS-015	Reliance on third-parties	 Low

## Reliance on third-parties

Interaction between smart contracts with third-party protocols like Uniswap and Pancakeswap. The audit's scope presupposes that third party entities will perform as intended and treats them as if they were black boxes. In the real world, third parties can be hacked and used against you. Additionally, improvements made by third parties may have negative effects, such as higher transaction costs or the deprecation of older routers.

## Recommendation

Regularly check third-party dependencies, and when required, reduce severe effects.

Error Code	Description	Severity
SLT: 078	Conformance to numeric notation best practices	● Low

## Too many digits

Literals with many digits are difficult to read and review.

```
_mint(msg.sender, 100000000 * 10 ** decimals()); //100 MILLION CKM TOKENS
```

## Recommendation

Use: [Ether suffix](#), [Time suffix](#), or [The scientific notation](#)

## Exploit Scenario

While 1\_ether looks like 1 ether, it is 10 ether. As a result, it's likely to be used incorrectly.

```
contract MyContract{
    uint 1_ether = 10000000000000000000;
}
```

Error Code	Description	Severity
SWC-104	CWE-252: Unchecked Return Value	● Low

## Unchecked Call Return Value

The return value of a message call is not checked. Execution will resume even if the called contract throws an exception. If the call fails accidentally or an attacker forces the call to fail, this may cause unexpected behaviour in the subsequent program logic.

```
function claimStuckTokens(address _token) external onlyOwner {
    if (_token == address(0x0)) {
        payable(owner()).transfer(address(this).balance);
        return;
    }
    IBEP20 BEP20token = IBEP20(_token);
    uint256 balance = BEP20token.balanceOf(address(this));
    BEP20token.transfer(owner(), balance);
}
```

## Recommendation

If you choose to use low-level call methods, make sure to handle the possibility that the call will fail by checking the return value.

Error Code	Description	Severity
SLT-054	Missing events arithmetic	 Low

## Missing events arithmetic


Missing events for critical arithmetic parameters.

```
function setBurnFeePercent (uint256 burn) external onlyOwner {  
    require(burn <= 10, "BEP20: burn fees should be less than equal to 10 percent");  
    _burnFee = burn;  
}
```

## Recommendation

Emit an event for critical parameter changes.



Error Code	Description	Severity
SWC-108	State variable visibility is not set	 Low

## State variable visibility is not set

Labeling the visibility explicitly makes it easier to catch incorrect assumptions about who can access the variable.

```
uint256 burnFee = 2;  
mapping(address =>bool) isWhitelisted;
```

## Recommendation

It is best practice to set the visibility of state variables explicitly. The default visibility for "isWhitelisted" is internal. Other possible visibility settings are public and private.

It is best practice to set the visibility of state variables explicitly. The default visibility for "burnFee" is internal. Other possible visibility settings are public and private.

## Centralization: Trading fee limitation

Error Code	Description
CEN-01	Check if the owner can set fees to an arbitrary high value

Coinsult tests if the owner of the smart contract can set the transfer, buy or sell fee to 25% or more. It is bad practice to set the fees to 25% or more, because owners can prevent healthy trading or even stop trading when the fees are set too high.

Finding	Status
Owner can <u>not</u> set fees to arbitrary high values	<span style="color: green;">●</span> Passed


Type of fee	Maximum fee
Trading fee	10
Buy fee	10
Sell fee	10

```
function setBurnFeePercent (uint256 burn) external onlyOwner {
    require(burn <= 10, "BEP20: burn fees should be less than equal to 10 percent");
    _burnFee = burn;
}
```

## Centralization: Pause trading

Error Code	Description
CEN-02	Check if the owner of the smart contract can pause / prevent trading


Coinsult tests if the owner of the smart contract has the ability to pause the contract. If this is the case, users can no longer interact with the smart contract; users can no longer trade the token.

Finding	Status
Owner can <u>not</u> pause trading	 Passed

## Centralization: Max transaction amount

Error Code	Description
CEN-03	Check if the contract implemented a system to prevent big transactions

Coinsult tests if the smart contract has a maximum transaction amount. If the transaction amount exceeds this limit, the transaction will revert. Owners could prevent normal transactions to take place if they abuse this function.

Finding	Status
Contract does <u>not</u> contain a maximum transaction amount	 Passed

## Centralization: Excluding from fees

Error Code	Description
CEN-04	Check if the owner can exclude addresses from trading fees

Coinsult tests if the owner of the smart contract can exclude addresses from paying tax fees. If the owner of the smart contract can exclude from fees, they could set high tax fees and exclude themselves from fees and benefit from 0% trading fees. However, some smart contracts require this function to exclude routers, dex, cex or other contracts / wallets from fees.

Finding	Status
Owner <u>can</u> exclude addresses from fees	<span style="color: yellow;">●</span> Pending

### Code to exclude from fees

```
function addWhitelistAddress (address user) external onlyOwner {
    require(user!= address(0), "can't whitelist zero address");
    isWhitelisted[user] = true;
}
```

## Centralization: Mint after deployment

Error Code	Description
CEN-05	Check if the owner can mint additional tokens after deployment

Coinsult tests if the owner of the smart contract can mint new tokens. If the contract contains a mint function, we refer to the token's total supply as non-fixed, allowing the token owner to "mint" more tokens whenever they want.

A mint function in the smart contract allows minting tokens at a later stage. A method to disable minting can also be added to stop the minting process irreversibly.

Minting tokens is done by sending a transaction that creates new tokens inside of the token smart contract. With the help of the smart contract function, an unlimited number of tokens can be created without spending additional energy or money.

Finding	Status
Owner can <u>not</u> mint new tokens after deployment	<span style="color: green;">●</span> Passed

## Centralization: Ability to blacklist

Error Code	Description
CEN-06	Check if the owner can blacklist addresses from trading

Coinsult tests if the owner of the smart contract can blacklist accounts from interacting with the smart contract. Blacklisting methods allow the contract owner to enter wallet addresses which are not allowed to interact with the smart contract.

This method can be abused by token owners to prevent certain / all holders from trading the token. However, blacklists might be good for tokens that want to rule out certain addresses from interacting with a smart contract.

Finding	Status
Owner can <u>not</u> blacklist addresses from trading	<span>●</span> Passed

## Centralization Privileges

The owner of the contract is allowed to interact with the following functions:

```
addWhitelistAddress()  
approve()  
claimStuckTokens()  
decreaseAllowance()  
increaseAllowance()  
removeWhitelistAddress()  
renounceOwnership()  
setBurnFeePercent()  
transfer()  
transferFrom()  
transferOwnership()
```



# Notes

## Notes by CMF

No notes provided by the team.

## Notes by Coinsult

No notes provided by Coinsult

# Constructor Snapshot

This is how the constructor of the contract looked at the time of auditing the smart contract.

```
constructor() BEP20("Check Me", "CKM") {  
    _mint(msg.sender, 100000000 * 10 ** decimals()); //100 MILLION CKM TOKENS  
    isWhitelisted[msg.sender] = true;  
}
```

# Disclaimer

This audit report has been prepared by Coinsult's experts at the request of the client. In this audit, the results of the static analysis and the manual code review will be presented. The purpose of the audit is to see if the functions work as intended, and to identify potential security issues within the smart contract.

The information in this report should be used to understand the risks associated with the smart contract. This report can be used as a guide for the development team on how the contract could possibly be improved by remediating the issues that were identified.

Coinsult is not responsible if a project turns out to be a scam, rug-pull or honeypot. We only provide a detailed analysis for your own research.

Coinsult is not responsible for any financial losses. Nothing in this contract audit is financial advice, please do your own research.

The information provided in this audit is for informational purposes only and should not be considered investment advice. Coinsult does not endorse, recommend, support or suggest to invest in any project.

Coinsult can not be held responsible for when a project turns out to be a rug-pull, honeypot or scam.

End of report

# Smart Contract Audit

November 10, 2022



Audit requested by

**Check Me Finance**

0xd1b5c3031c1b107fa7db2dfba14eba83a25d0202