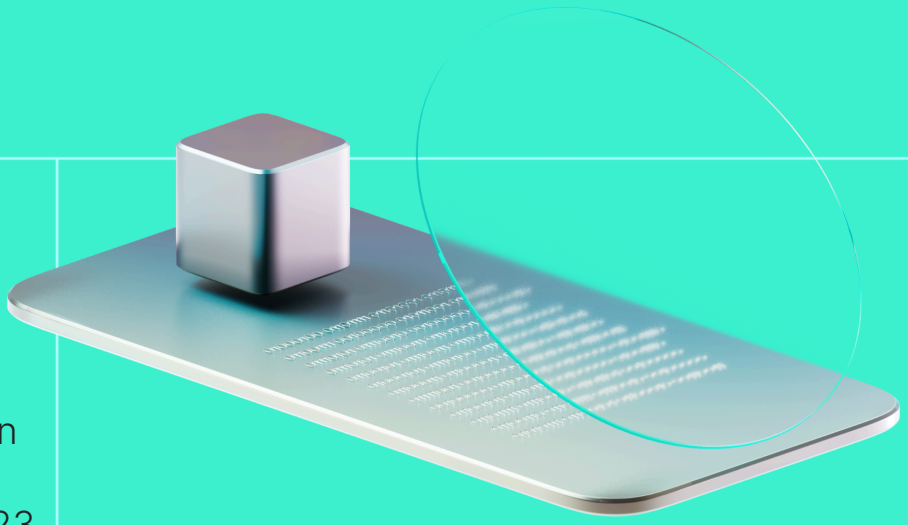
HACKEN

Smart Contract Code Review And Security Analysis Report



Customer: Wooooo Coin

Date: 05 December, 2023

Hacken OU
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia

We thank Wooooo Coin for allowing us to conduct a Smart Contract Security Assessment. This document outlines our methodology, limitations, and results of the security assessment.

Wooooo Coin is an innovative ERC20 token. It integrates advanced functionalities like auto-liquidity, dynamic liquidity levels, and a distinct fee system for various transaction types, all while embracing the flamboyant essence of its wrestling legend namesake.

Platform: EVM

Language: Solidity

Tags: ERC20

Timeline: 16.11.2023 - 21.11.2023

Methodology: Link

Last review scope

Repository	https://github.com/yodolph/Wooooo
Commit	5ddd0c2

View full scope

Hacken OU
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia

Audit Summary

10/10	10/10	0%	10/10
Security score	Code quality score	Test coverage	Documentation quality score

Total: 3/10

The system users should acknowledge all the risks summed up in the risks section of the report.

8	7	0	1
Total Findings	Resolved	Acknowledged	Mitigated

Findings by severity	Findings Number	Resolved	Mitigated	Acknowledged
Critical	0	0	0	0
High	1	1	0	0
Medium	3	2	1	0
Low	5	5	0	0

Hacken OU
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia

Document

Name	Smart Contract Code Review and Security Analysis Report for Wooooo Coin
Approved By	Kaan Caglan | Senior SC Auditor at Hacken OÜ
Website	N/A
Changelog	21.11.2023 – Preliminary Report 05.12.2023 – Final Report

Introduction

Hacken OÜ (Consultant) was contracted by Wooooo Coin (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report presents the findings of the security assessment of the Customer's smart contracts.

System Overview

Wooooo coin is an innovative ERC20 token with the following contract:

- WooooCoin - A contract that encapsulates various features for liquidity management, fee structuring, and investor engagement. It integrates auto-liquidity provisions, dynamic liquidity levels, and a detailed fee system for different transaction types, all built on the robust OpenZeppelin ERC20 standard.

Privileged roles

The Wooooo Coin contract, leveraging OpenZeppelin's Ownable standard, ensures secure and restricted access to critical functions. The owner has exclusive rights to:

- Set transaction fees for buying, selling, and transfers.
- Adjust liquidity thresholds and operational parameters.
- Manage vested tokens and associated payouts.
- Enable or disable key features like auto-liquidity and treasury contributions.

Executive Summary

The score measurement details can be found in the corresponding section of the scoring methodology.

Documentation quality

The total Documentation Quality score is **10** out of **10**.

- NatSpec is sufficient.
- Technical description is provided.

Code quality

The total Code Quality score is **10** out of **10**.

- Best practices are followed.
- The code is structured and readable.

Test coverage

Code coverage of the project is **0%** (branch coverage), with a mutation score of 0%.

- Coverage tool could not be run because there is not any test case.

Security score

As a result of the audit, the code does not contain any severity issues. The security score is **10** out of **10**.

All found issues are displayed in the “Findings” section.

Summary

According to the assessment, the Customer's smart contract has the following score: **3.0**.

The system users should acknowledge all the risks summed up in the risks section of the report.

Risks

- The contract's design grants the owner extensive control over its functions, including financial operations. This centralization poses a significant risk, where the owner could potentially withdraw all assets, jeopardizing user investments and trust. Such control undermines the decentralization principle in blockchain, increasing the risk of misuse or mismanagement.

- If a malicious actor obtains tokens before **LP_STATE** is set to true (indicating readiness for transfers), they can exploit the _transfer function's design. In scenarios where **LP_STATE** is false and neither the sender nor the recipient is exempt, the function adds the recipient to the deny list. A malicious actor, having acquired tokens prematurely, could initiate minimal token transfers to various addresses. This would lead to the unjust addition of these addresses to the deny list, effectively blocking their ability to participate in future transfers and potentially causing a widespread denial of service within the contract's ecosystem

- The implementation of the liquidation function in your smart contract creates a systemic risk of unbalancing the token and native token (e.g., BNB) reserves in the liquidity pool. Each time this function is called, it swaps a significant portion of tokens (938 out of 1000 parts) for the native token, and then only a smaller fraction (62 out of 1000 parts) of these tokens is paired back into the liquidity pool. This repeated process incrementally increases the token reserve and decreases the native token reserve in the pool. And it means the token price will decrease in time.

- The Wooooo function, not utilized within the contract but relying on **block.timestamp** for generating pseudo-randomness poses a risk of predictability, making it unsuitable for crucial randomness-dependent tasks. This predictability could lead to manipulation or exploitation, especially if used in critical applications off-chain.
- sortReserves function will work smoothly as long as there will be more USDC reserve than BNB and more WOO Token reserve than USDC in the uniswap. Otherwise it will return in the wrong order and it will affect token price.
- The contract's liquidity addition function is at risk of integer underflow if the PAIR_TOKENS exceed the contract's initial balance. This issue may cause the function to revert due to underflow, leading to failed transactions when attempting to calculate liquidity values. While this reversion prevents incorrect liquidity entries, it also indicates a vulnerability in handling larger token pairings than the contract's balance, potentially affecting its operational smoothness and user trust.

Findings

Critical

No critical severity issues were found.

High

H01. Incorrect Reserve Order in priceBNB Function Leading to Erroneous Price Calculations

Impact	High
Likelihood	Medium

The priceBNB function assumes that the first element returned by getReserves is always the reserve for USDC. This assumption holds true as long as the USDC address is smaller than the BNB address, ensuring that USDC appears first in the pair. However, the contract includes a setCurrency function that allows the contract owner to change the currency to another stablecoin like BUSD. Since the BUSD address is larger than the BNB address, changing to BUSD would result in incorrect reserve order in getReserves. This would lead to miscalculations in the priceBNB function, significantly impacting the accuracy of the priceUSD calculation and potentially causing substantial pricing errors in the contract's operations.

```
 function priceInBNB() public view returns (uint256) {
       (uint256 reserveToken, uint256 reserveBNB,) =
IPair(pair).getReserves();
       return (reserveBNB* 1e18)/reserveToken;
   }
```

Path: ./wooooo-coin-bnb-final.sol*:*

POC:

```
 uint256 tokenPriceWithUSDC = woooCoin.priceUSD();
       vm.startPrank(deployer);
       CURRENCY = address(0xe9e7CEA3DedcA5984780Bafc599bD69ADd087D56); //
BUSD
       woooCoin.setCurrency(CURRENCY);
       vm.stopPrank();
       uint256 tokenPriceWithBUSD = woooCoin.priceUSD();
       console.log("Token price with USDC : %s, token price with BUSD:
%s",tokenPriceWithUSDC,  tokenPriceWithBUSD);
```

Output:

```
Token price with USDC : 785835285899290864605, token price with
BUSD: 11584371928675559
```

Recommendation: To correct this issue and ensure that the first parameter in the getReserves call is always USDC (or the equivalent stablecoin), the priceBNB function should be modified to dynamically identify and correctly assign the reserve values based on the addresses of the tokens involved. This can be achieved by comparing the address of the stablecoin (e.g., USDC or BUSD) with the address of BNB, and then accordingly assigning the reserve values. Here is an example of how this can be implemented:

```
function priceBNB() public view returns (uint256) {
```

```
    (uint256 reserve0, uint256 reserve1,) =
IPair(currency).getReserves();
    uint256 reserveUSDC;
    uint256 reserveBNB;
    // Check which reserve is USDC/BUSD and which is BNB
    if (CURRENCY < PAIR) {
        reserveUSDC = reserve0;
        reserveBNB = reserve1;
    } else {
        reserveUSDC = reserve1;
        reserveBNB = reserve0;
    }

    return (reserveUSDC * 1e18) / reserveBNB;
}
```

This modification ensures that regardless of the address sizes of the stablecoin and BNB, the reserves are correctly identified and used in calculations, maintaining the accuracy of the priceUSD function.

Found in: 5ddd0c2

Status: Fixed (Revised file sha3sum: 8d5bea234ed2f294c85eb7aaece9e9c86688c704b209abc13dc38b44)

Remediation: Client has introduced a new function named `sortReserves`.

Medium

M01. Underflow Error In Liquidity Addition Process

Impact	Medium
Likelihood	Low

The Solidity code for a liquidity addition process is vulnerable to an integer underflow error. This occurs when *PAIR_TOKENS*, the estimated tokens to be paired with LP Tokens, exceeds the initial balance of the contract. In such a scenario, the calculation of the final amount (i.e., *uint256 amount = address(this).balance - current;*) will result in an underflow error, as it attempts to subtract a larger number from a smaller one. This flaw arises due to the calculation of current and amount without considering the potential for *PAIR_TOKENS* to be greater than the initial balance.

Path：./wooooo-coin-bnb-final.sol*:*

Recommendation: To mitigate this issue, a validation check should be introduced before the liquidity addition process. This check should ensure that the initial balance is greater than future possible PAIR_TOKENS. If this condition is not met, the operation should not proceed. Implementing this check will prevent the scenario where the subtraction results in an underflow. An example implementation is:

```
uint256 initial = address(this).balance;
```

```
uint256 LP_TOKENS = (tokens * 62) / 1000;
uint256 PAIR_TOKENS = estimateEthOutAfterLiquify(LP_TOKENS);
//Estimate Tokens to be paired with LP Tokenswith the amount of
token after liquify
if(impact < impactLevel && initial > PAIR_TOKENS) {
    // Continue with the liquidity addition process
}
```

Found in: 5ddd0c2

Status: Mitigated (Revised file sha3sum: 8d5bea234ed2f294c85eb7aaece9e9c86688c704b209abc13dc38b44)

Remediation: Client ensures that, initial balance value will always be bigger than the possible PAIR_TOKENs amount. Due to this no control is required.

M02. Incorrect Reserve Token Order Assumption

Impact	High
Likelihood	Low

The function priceInBNB in the contract assumes that the first element returned by the getReserves function of the Uniswap pair (IPair(pair).getReserves()) is always the reserve of the Wooooo Coin token (reserveToken). This assumption is potentially flawed because the order of reserves returned by getReserves depends on the sorting of the token addresses in the pair. If the Wooooo Coin token address is greater than the address of the paired token (such as WETH or BNB), this assumption will lead to an incorrect calculation of the price in BNB.

```
 function priceInBNB() public view returns (uint256) {
       (uint256 reserveToken, uint256 reserveBNB,) =
IPair(pair).getReserves();
       return (reserveBNB* 1e18)/reserveToken;
   }
```

Path: ./wooooo-coin-bnb-final.sol*:*

Recommendation: To mitigate this issue:

- Contract Initialization Check: Implement a validation check in the contract's constructor to ensure that the contract is not initialized if the Wooooo Coin token address is higher than the WETH/BNB address. This approach prevents the deployment of the contract in a configuration that could lead to incorrect price calculations.
- Adjust priceInBNB Function: Modify the priceInBNB function to dynamically determine the correct order of reserves based on the token addresses. This ensures the function accurately identifies which reserve corresponds to the Wooooo Coin token and which to BNB, regardless of their address order.

```
function priceInBNB() public view returns (uint256) {
    (uint256 reserve0, uint256 reserve1,) =
IPair(pair).getReserves();
    (uint256 reserveToken, uint256 reserveBNB) = address(this) <
pair ? (reserve0, reserve1) : (reserve1, reserve0);
    return (reserveBNB * 1e18) / reserveToken;
}
```

Found in: 5ddd0c2

Status: Fixed (Revised file sha3sum: 8d5bea234ed2f294c85eb7aaece9e9c86688c704b209abc13dc38b44)

Remediation: Client has introduced a new function named `sortReserves`.

M03. Highly Centralized Functionality

Impact	Medium
Likelihood	Medium

The addInvestor function in its current form allows the contract owner to add investors and allow them to receive treasury tokens at any time. This presents a significant centralization risk, as the owner has unilateral control over the distribution of tokens. Such centralization goes against the principles of decentralization and fairness in blockchain systems. It could lead to potential misuse or favoritism, where the owner might add investors selectively or in a biased manner.

```
   function addInvestor(address addr, uint256 tokens)external
onlyOwner{
        vested[addr]=tokens;
        vestedKey.push(addr);
    }
```

Path：./wooooo-coin-bnb-final.sol*:*

Recommendation: To mitigate this risk, it is recommended to implement a mechanism that restricts the addition of new investors after a certain milestone, such as the completion of the airdrop.

Found in: 5ddd0c2

Status: Fixed (Revised file sha3sum: 8d5bea234ed2f294c85eb7aaece9e9c86688c704b209abc13dc38b44)

Remediation: Client has added a `require(!LP_STATE)` control to make sure the owner won't be able to add a new investor after airdrop.

Low

L01. Missing Zero Address Validation

Impact	Medium
Likelihood	Low

The smart contract does not validate for the zero address (0×0) when handling address parameters. This oversight could inadvertently trigger unintended external calls to the 0×0 address, which might lead to undesired behaviors or potential loss of funds.

Path：./wooooo-coin-bnb-final.sol*:*

```
494:        txn.marketing = addr;
497:        txn.WoooooEnergy = addr;
500:        txn.RicFlair = addr;
503:        txn.network = addr;
530:        payouts[investor]=addr;
540:        CURRENCY = addr;
```

Recommendation：To safeguard against unintended interactions with the zero address, it is advised to integrate the following best practices:

1. Validation Checks: Implement validation checks at the start of functions or operations that involve address parameters. These checks should confirm that the address is not the zero address (0×0) before proceeding with further execution.
2. Reusable Modifier: Consider creating a reusable modifier such as isNotZeroAddress(address _address), which can be applied to functions to ensure that they are not passed or dealing with a zero address. This not only enhances code reusability but improves clarity.
3. Error Handling: If an address validation fails, ensure that the contract emits a clear and meaningful error message. This assists in debugging and alerts users to potential issues with their transactions.
4. Testing: After implementing the above changes, it is crucial to conduct comprehensive testing to ensure the smart contract behaves as expected and does not interact with the zero address.

By adhering to these recommendations, it is possible to reduce the risk associated with unintended external calls to the 0×0 address and enhance the robustness of smart contracts.

Found in： 5ddd0c2

Status: Fixed (Revised file sha3sum:

8d5bea234ed2f294c85eb7aaece9e9c86688c704b209abc13dc38b44)

Remediation: Client has added a modifier named `validAdress`.

L02. Use of transfer or send Instead of call To Send Native Assets

Impact	Medium
Likelihood	Low

The use of transfer() in the contracts may lead to unintended outcomes for the native asset being sent to the receiver. The transaction will fail under the following circumstances:

- The receiver address is a smart contract that does not implement a payable function.
- The receiver address is a smart contract that implements a payable fallback function using more than 2300 Gas units.
- The receiver address is a smart contract that implements a payable fallback function requiring less than 2300 Gas units but is called through a proxy, causing the call's Gas usage to exceed 2300.
- In addition, using a Gas value higher than 2300 might be mandatory for certain multi-signature wallets.

Path： ./wooooo-coin-bnb-final.sol*:*

```
payable(payouts[investor]).transfer(payment);

payable(investor).transfer(payment);

payable(txn.marketing).transfer((amount*ops.marketing)/1000);

payable(txn.WoooooEnergy).transfer((amount*ops.WoooooEnergy)/1000);
```

```
payable(txn.RicFlair).transfer((amount*ops.RicFlair)/1000);

payable(txn.network).transfer((amount*ops.network)/1000);

payable(owner()).transfer(amount);

payable(msg.sender).transfer(amount);
```

Recommendation: Use call() function instead of transfer() for the native token transfers.

Found in: 5ddd0c2

Status: Fixed (Revised file sha3sum:

86c25361784d3b5f098c9157ff733edfa78d7664fa17b5d51e80b820)

L03. Incorrect Vested Token Tracking Leading to Operational Inefficiencies

Impact	Medium
Likelihood	Low

The function vestedSell is designed to track the sale of vested tokens by an investor. However, there is a logical flaw in its implementation. The function increases *sold[seller]* by amount and checks if *sold[seller]* equals *vested[seller]*. The issue arises when an investor acquires additional tokens from external sources and sells them. In this scenario, *sold[seller]* can exceed *vested[seller]*, resulting in a condition where the *soldout[seller]* flag is never set to true, and the ops variable is not updated as intended. This flaw can prevent the correct execution of key operational logic dependent on the ops variable.

Path: ./wooooo-coin-bnb-final.sol*:*

Recommendation: To address this issue, it is recommended to restrict investors from receiving or buying additional tokens beyond their vested amount. This can be achieved by adding checks in the token transfer logic to ensure that only vested investors can receive or buy tokens. These checks will enforce that only tokens from the vested pool are sold, ensuring the accuracy of *sold[seller]*.

Found in: 5ddd0c2

Status: Fixed (Revised file sha3sum: 8d5bea234ed2f294c85eb7aaece9e9c86688c704b209abc13dc38b44)

Remediation: Client solved this issue with adding isInvestor control to _transfer function under `else if(sender == pair)` and `else{` parts.

L04. Rigid Full-Sale Disqualification Threshold in vestedSell Function

Impact	Medium
Likelihood	Medium

In the current implementation of the vestedSell function, an investor is marked as 'sold out' and becomes ineligible for treasury fees as soon as they sell 100% of their vested tokens. This all-or-nothing approach might be overly strict, potentially penalizing investors who have sold a significant portion but not all of their tokens. This could discourage active trading or selling of tokens, as investors might be reluctant to sell their holdings completely due to fear of losing all treasury benefits.

```
function vestedSell(address seller,uint256 amount)private{
        sold[seller]+=amount; //Track vested tokens being sold
        if(sold[seller]==vested[seller]){
            soldout[seller]=true;
            out+=1;
            if(out>=vestedKey.length){
                ops = OPS( 0, 500, 125, 125, 250 );
            }
        }
    }
```

Path: ./wooooo-coin-bnb-final.sol:

Recommendation: To create a more balanced and incentivized structure, it's advisable to introduce a threshold less than 100% for marking an investor as 'sold out'. For instance, you could set a threshold where an investor is considered 'sold out' if they sell a certain percentage (e.g., 80%) of their vested tokens. This threshold can be determined based on the contract's economic model and desired investor behavior. Adjusting the threshold allows for more flexibility and can encourage investors to remain active in the market without the fear of immediate total disqualification from treasury fees. This modification would involve changing the condition in the if statement to check whether sold[seller] is greater than or equal to a certain percentage of vested[seller]. For example:

```
uint256 thresholdPercentage = 80; // Example percentage
uint256 thresholdAmount = vested[seller] * thresholdPercentage /
100;
if (sold[seller] >= thresholdAmount) {
    soldout[seller] = true;
    // ... rest of the code
}
```

Found in: 5ddd0c2

Status: Fixed (Revised file sha3sum:

8d5bea234ed2f294c85eb7aaece9e9c86688c704b209abc13dc38b44)

Remediation: Client solved this issue with adding *vestedMin* control.

```
for (uint32 i; i < len;) {
                    uint256 vestedMin = vested[vestedKey[i]]/4;
                    uint256 bal = balanceOf(vestedKey[i]);
                    if(vestedKey[i]!=seller && !soldout[vestedKey[i]] &&
bal>=vestedMin){
```

L05. Hardcoded Zero Slippage in Liquidity Operations

Impact	Medium
Likelihood	Low

In the specified contract, particularly within the functions for adding liquidity (addLiquidityETH) and swapping tokens (swapExactTokensForETHSupportingFeeOnTransferTokens), a hardcoded value of 0 is used for the slippage parameter. This presents a potential issue as zero slippage is often unrealistic in a live trading environment. Hardcoded slippage values can lead to failed transactions in cases where the actual price movement exceeds the specified slippage limit, especially in volatile market conditions. This can result in an inability to execute swaps or add liquidity at critical times, potentially impacting the contract's effectiveness and user experience.

```
340:                router.addLiquidityETH{ value: _pair
}(address(this), _tokens, 0, 0, owner(), block.timestamp);  //
@audit-issue

355:
router.swapExactTokensForETHSupportingFeeOnTransferTokens(
356:            TOKENS,
357:            0,    // @audit-issue
358:            path,
359:            address(this),
360:            block.timestamp
361:        );
```

Path：./wooooo-coin-bnb-final.sol*:*

Recommendation: To resolve the issue of hardcoded zero slippage in liquidity operations, the contract should incorporate three state variables: *uint256 minSwapAmount, uint256 minLiquidityAddNativeAmount, and uint256 minLiquidityAddTokenAmount*. These variables will represent minimum acceptable amounts for swapping and adding liquidity for both native and token assets. A setter function, accessible only to the contract owner, can be implemented to adjust these values as needed. The *swapExactTokensForETHSupportingFeeOnTransferTokens* function will utilize *minSwapAmount*, while *addLiquidityETH* will use *minLiquidityAddNativeAmount* and *minLiquidityAddTokenAmount*. This approach allows for more precise and adaptable control over slippage in different transaction types, enhancing the contract's flexibility and reducing the risk of failed transactions due to market volatility.

Found in: 5ddd0c2

Status: Fixed (Revised file sha3sum: 86c25361784d3b5f098c9157ff733edfa78d7664fa17b5d51e80b820)

Informational

I01. Ownership Irrevocability Vulnerability in Smart Contract

The smart contract under inspection inherits from the *Ownable* library, which provides basic authorization control functions, simplifying the implementation of user permissions. Given this, once the owner renounces ownership using the renounceOwnership function, the contract becomes ownerless. As evidenced in the provided transaction logs, after the renounceOwnership function is called, attempts to call functions that require owner permissions fail with the error message: "Ownable: caller is not the owner."

This state renders the contract's adjustable parameters immutable and potentially makes the contract useless for any future administrative changes that might be necessary.

Path：./wooooo-coin-bnb-final.sol*:*

```
//ERC20 is the same as BEP20
contract WoooooCoin is ERC20, Ownable, ReentrancyGuard  {
```

Recommendation: To mitigate this vulnerability:

1. Override the renounceOwnership function to revert transactions: By overriding this function to simply revert any transaction, it will become impossible for the contract owner to unintentionally (or intentionally) render the contract ownerless and thus immutable.
2. Implement an ownership transfer function: While the Ownable library does provide a transferOwnership function, if this is not present or has been removed from the current contract, it should be re-implemented to ensure there is a way to transfer ownership in future scenarios.

Found in: 5ddd0c2

Status: Fixed (Revised file sha3sum:

8d5bea234ed2f294c85eb7aaece9e9c86688c704b209abc13dc38b44)

I02. Avoid Unnecessary Initializations Of Uint256 And Bool Variable To 0/false

In Solidity, it is common practice to initialize variables with default values when declaring them. However, initializing `uint256` variables to `0` and `bool`variables to `false` when they are not subsequently used in the code can lead to unnecessary Gas consumption and code clutter. This issue points out instances where such initializations are present but serve no functional purpose.

Path: ./wooooo-coin-bnb-final.sol*:*

```
412:                uint256 totalShares = 0;

414:                for (uint256 i = 0; i < vestedKey.length; i++) {

420:                for (uint256 i = 0; i < vestedKey.length; i++) {
```

Recommendation: It is recommended not to initialize integer variables to 0 to and boolean variables to false to save some Gas.

Found in: 5ddd0c2

Status: Fixed (Revised file sha3sum:

8d5bea234ed2f294c85eb7aaece9e9c86688c704b209abc13dc38b44)

I03. Custom Errors For Better Gas Efficiency

Using custom errors instead of revert strings can significantly reduce Gas costs, especially when deploying contracts. Prior to Solidity v0.8.4, revert strings were the only way to provide more information to users about why an operation failed. However, revert strings are expensive, and it is difficult to use dynamic information in them. Custom errors, on the other hand, were introduced in Solidity v0.8.4 and provide a gas-efficient way to explain why an operation failed.

Path: ./wooooo-coin-bnb-final.sol:

```
require(_address != address(0), "Invalid address");
require(fees.treasury + fees.lp <= 250, "Cannot exceed 25%");
require(amount != 0, "Must.Not.Be.Zero");
require(!deny[recipient], "Snipe.Attacker.Not.Permitted");
require(enabled, "Trading.Disabled");
require(amount <= max.buy, "Amount.Buy.Exceeded");
require(balanceOf(recipient) + amount <= max.addr, "Balance.Exceeded");
require(amount <= max.sell, "Amount.Sell.Exceeded");
require(balanceOf(recipient) + amount <= max.addr, "Balance.Exceeded");
require(!LP_STATE && !enabled && !open, "Trading.Already.Launched");
```

Recommendation: It is recommended to use custom errors instead of revert strings to reduce Gas costs, especially during contract deployment. Custom errors can be defined using the error keyword and can include dynamic information.

Found in: 5ddd0c2

Status: Acknowledged

I04. Cache State Variable Array Length In For Loop

Failing to cache the array length when iterating through arrays in Solidity can have significant performance and Gas cost implications. In Solidity, array lengths can change during execution due to external calls or storage modifications. When the array length is not cached before entering a loop, it is recomputed with each iteration, leading to unnecessary Gas consumption and potentially making the contract vulnerable to reentrancy attacks.

Path: ./wooooo-coin-bnb-final.sol*:*

```
414:                for (uint256 i = 0; i < vestedKey.length; i++) {

420:                for (uint256 i = 0; i < vestedKey.length; i++) {
```

Recommendation: Accessing a state variable like vestedKey directly within a loop in a smart contract can lead to efficiency and cost issues, especially if the array grows large. Each read operation from a state variable in Ethereum consumes Gas, and in a loop, this can quickly become expensive. Instead of accessing vestedKey.length directly in the loop condition, store it in a local variable. This reduces the number of state reads.

```
uint256 length = vestedKey.length;
for (uint256 i = 0; i < length; i++) {
    // ...
}
```

Found in: 5ddd0c2

Status: Fixed (Revised file sha3sum:

8d5bea234ed2f294c85eb7aaece9e9c86688c704b209abc13dc38b44)

I05. Immutable Keyword For Gas Optimization

There are variables that do not change, so they can be marked as immutable to greatly improve the Gas costs.

Path: ./wooooo-coin-bnb-final.sol*:*

```
64:    uint256 public supply;
88:    IUniswapV2Router02 public router;

89:    address public pair;

91:    address public ROUTER;

92:    address public PAIR;
```

Recommendation: Consider marking state variables as an immutable that never changes on the contract.

Found in: 5ddd0c2

Status: Fixed (Revised file sha3sum:

8d5bea234ed2f294c85eb7aaece9e9c86688c704b209abc13dc38b44)

Remediation: Changed to immutable variables that can be changed.

I06. Unused State Variable

There are state variables which are declared but not used in any function. This might increase the contract's Gas usage unnecessarily.

Path: ./wooooo-coin-bnb-final.sol*:*

```
address public treasury
```

Recommendation: Remove redundant variables to save Gas on deployment and increase code quality.

Found in: 5ddd0c2

Status: Fixed (Revised file sha3sum:

8d5bea234ed2f294c85eb7aaece9e9c86688c704b209abc13dc38b44)

I07. Increments Can Be 'unchecked' In For Loops

Newer versions of the Solidity compiler will check for integer overflows and underflows automatically. This provides safety but increases Gas costs.

When an unsigned integer is guaranteed to never overflow, the unchecked feature of Solidity can be used to save Gas costs.

A common case for this is for-loops using a strictly-less-than comparison in their conditional statement, e.g.:

```
uint256 length = someArray.length;
for (uint256 i; i < length; ++i) {
}
```

In cases like this, the maximum value for length is 2**256 - 1. Therefore, the maximum value of i is 2**256 - 2 as it will always be strictly less than length.

This example can be replaced with the following construction to reduce Gas costs:

```
for (uint i; i < length; ) {
    // do something that doesn't change the value of i
    unchecked {
        ++i;
    }
}
```

Path: ./wooooo-coin-bnb-final.sol*:*

```
414:                for (uint256 i = 0; i < vestedKey.length; i++) {
420:                for (uint256 i = 0; i < vestedKey.length; i++) {
```

Recommendation: Use unchecked math to block overflow / underflow check to save Gas.

Found in: 5ddd0c2

Status: Fixed (Revised file sha3sum: 8d5bea234ed2f294c85eb7aaece9e9c86688c704b209abc13dc38b44)

I08. Style Guide Violation

The provided projects should follow the official guidelines.

Inside each contract, library or interface, use the following order:

1. *Type declarations*
2. *State variables*
3. *Events*
4. *Modifiers*
5. *Functions*

Functions should be grouped according to their visibility and ordered:

1. *constructor*
2. *receive function (if exists)*
3. *fallback function (if exists)*
4. *external*
5. *public*
6. *internal*
7. *private*

Within a grouping, place the view and pure functions last.

It is best practice to follow the Solidity naming convention. This will increase overall code quality and readability.

Path: ./*

Recommendation: follow the official Solidity guidelines.

Found in: 5ddd0c2

Status: Fixed (Revised file sha3sum: 8d5bea234ed2f294c85eb7aaece9e9c86688c704b209abc13dc38b44)

I09. Redundant State Variables in launch Function

The launch function in smart contract sets three state variables - enabled, open, and LP_STATE - to TRUE. All these variables are initially FALSE and lack distinct methods for individual state changes. This redundancy is inefficient as each variable update consumes Gas, and they all represent the same operational state of the contract. This design leads to unnecessary Gas usage and adds complexity without providing distinct functional benefits.

Path：./wooooo-coin-bnb-final.sol*:*

```
function launch()external onlyOwner{
     require(!LP_STATE && !enabled && !open, "Trading.Already.Launched");
     enabled = true;
     open = true;
     LP_STATE = true;
     emit WoooooEvent("Wooooo!");
 }
```

Recommendation: To improve Gas efficiency and simplify the contract's logic, consider consolidating these three state variables into a single variable that represents the operational state. This single variable could be named to clearly indicate the contract's state (e.g., isTradingActive or contractState). The launch function would then only need to update this one variable, significantly reducing gas costs associated with multiple state changes. Additionally, this change would make the contract's code cleaner and more maintainable, as it reduces the potential for errors or confusion arising from having multiple variables that serve the same purpose.

Found in: 5ddd0c2

Status: Fixed (Revised file sha3sum: 8d5bea234ed2f294c85eb7aaece9e9c86688c704b209abc13dc38b44)

I10. Redundant Trading State Check in _transfer Function

The _transfer function includes a redundant condition check *if(!exempt[sender] && !exempt[recipient]){require(enabled, "Trading.Disabled");}*. This check is unnecessary because the function is already gated by the *LP_STATE* being true, and the launch function simultaneously sets *LP_STATE*, enabled, and open to *TRUE*. Given that these variables are always updated together, the state represented by enabled is redundant when *LP_STATE* is already checked, leading to inefficiency in contract execution.

Path：./wooooo-coin-bnb-final.sol*:*

```
function _transfer(address sender,address recipient,uint256 amount) internal
override  validAddress(sender) validAddress(recipient){
        require(amount != 0, "Must.Not.Be.Zero");
        require(!deny[recipient], "Snipe.Attacker.Not.Permitted");
        if(LP_STATE){
            if(!exempt[sender] && !exempt[recipient]){require(enabled,
"Trading.Disabled");}
```

Recommendation: To streamline the *_transfer* function and reduce gas costs, it's recommended to remove the redundant check for the enabled state. Since *LP_STATE* adequately represents the trading readiness of the contract, the additional condition involving enabled is unnecessary. Simplifying this check will make the contract more gas-efficient and easier to understand, maintaining the necessary security and functional checks without the overhead of redundant state validations.

Found in: 5ddd0c2

Status: Fixed (Revised file sha3sum: 8d5bea234ed2f294c85eb7aaece9e9c86688c704b209abc13dc38b44)

I11. Overuse of Identical WoooooEvent Strings in Multiple Contract Functions

The WoooooEvent with the string "Wooooo!" is emitted in several different contexts within the contract, including in the *_transfer*, *constructor*, *launch*, and *setCurrency* functions. This repeated use of the same event string across multiple, functionally distinct parts of the contract can lead to confusion. It becomes challenging to distinguish the specific function that triggered the event when analyzing logs or debugging, as the event message does not provide context or specificity about its source or the nature of the state change.

Path: ./wooooo-coin-bnb-final.sol*:*

```
emit WoooooEvent("Wooooo!");
```

Recommendation: To enhance clarity and debugging efficacy, it is recommended to refactor the event messages associated with WoooooEvent emissions. Each event string should be unique and descriptive of the context in which it is emitted. For instance, in the launch function, the event could be "Launch: Trading Started", while in setCurrency, it could be "Currency Set: [Details]". These tailored event messages will provide clearer insights into contract behavior, aid in tracking the contract's activities, and improve overall maintainability.

Found in: 5ddd0c2

Status: Acknowledged

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with 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).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

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 recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Hacken OU
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia

Appendix 1. Severity Definitions

When auditing smart contracts Hacken is using a risk-based approach that considers the potential impact of any vulnerabilities and the likelihood of them being exploited. The matrix of impact and likelihood is a commonly used tool in risk management to help assess and prioritize risks.

The impact of a vulnerability refers to the potential harm that could result if it were to be exploited. For smart contracts, this could include the loss of funds or assets, unauthorized access or control, or reputational damage.

The likelihood of a vulnerability being exploited is determined by considering the likelihood of an attack occurring, the level of skill or resources required to exploit the vulnerability, and the presence of any mitigating controls that could reduce the likelihood of exploitation.

Risk Level	High Impact	Medium Impact	Low Impact
High Likelihood	Critical	High	Medium
Medium Likelihood	High	Medium	Low
Low Likelihood	Medium	Low	Low

Risk Levels

Critical: Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.

High: High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.

Medium: Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.

Low: Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution, do not affect security score but can affect code quality score.

Impact Levels

High Impact: Risks that have a high impact are associated with financial losses, reputational damage, or major alterations to contract state. High impact issues typically involve invalid calculations, denial of service, token supply manipulation, and data consistency, but are not limited to those categories.

Medium Impact: Risks that have a medium impact could result in financial losses, reputational damage, or minor contract state manipulation. These risks can also be associated with undocumented behavior or violations of requirements.

Low Impact: Risks that have a low impact cannot lead to financial losses or state manipulation. These risks are typically related to unscalable functionality, contradictions, inconsistent data, or major violations of best practices.

Likelihood Levels

High Likelihood: Risks that have a high likelihood are those that are expected to occur frequently or are very likely to occur. These risks could be the result of known vulnerabilities or weaknesses in the contract, or could be the result of external factors such as attacks or exploits targeting similar contracts.

Medium Likelihood: Risks that have a medium likelihood are those that are possible but not as likely to occur as those in the high likelihood category. These risks could be the result of less severe vulnerabilities or weaknesses in the contract, or could be the result of less targeted attacks or exploits.

Low Likelihood: Risks that have a low likelihood are those that are unlikely to occur, but still possible. These risks could be the result of very specific or complex vulnerabilities or weaknesses in the contract, or could be the result of highly targeted attacks or exploits.

Informational

Informational issues are mostly connected to violations of best practices, typos in code, violations of code style, and dead or redundant code.

Informational issues are not affecting the score, but addressing them will be beneficial for the project.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Initial review scope

Repository	https://github.com/yodolph/Wooooo
Commit	5ddd0c2
Whitepaper	Not provided
Requirements	Not provided
Technical Requirements	Not provided

Contracts in Scope

wooooo-coin-bnb-final.sol

Hacken OU
Parda 4, Kesklinn, Tallinn
10151 Harju Maakond, Eesti
Kesklinna, Estonia

Second review scope

Repository	Not provided
Commit	Not provided
Whitepaper	Not provided
Requirements	Not provided
Technical Requirements	Not provided

Contracts in Scope

File: wooooo-coin-bnb-final.sol
SHA3:
86c25361784d3b5f098c9157ff733edfa78d7664fa17b5d51e80b820