QuillAudits

Audit Report
February, 2022

For

Contents

Overview

Vanilla Bet by Vanilla Network

Vanilla Bet is a reflection BEP-20 token delivering powerful decentralized social gaming and staking dApps driven by Blockchain technology.

Scope of the Audit

The scope of this audit was to analyze the Vanilla Bet smart contract's codebase for quality, security, and correctness.

Contract code was taken from:
Version 1: https://testnet.bscscan.com/address/0x2438ac6c2dfc9c00013960eb5e04b9bdc133f317

Fixed In:
Version 2: https://testnet.bscscan.com/address/0x246975D1c2Df27653cdE8d169329E3938AbDc919

Time Duration - Jan 13, 2022 - Jan 20,2022

Mainnet Address:
https://bscscan.com/address/0x457cab4A14b259b84F99d58F1a99B11FBAb2044a#code

Checked Vulnerabilities

We have scanned the smart contract for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered:

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Malicious libraries
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- ERC20 transfer() does not return boolean
- ERC20 approve() race
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls
- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Using block.timestamp
- Multiple Sends
- Using SHA3
- Using suicide
- Using throw
- Using inline assembly

Techniques and Methods

Throughout the audit of the smart contract, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC-20/BEP20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods and tools were used to review all the smart contracts.

Structural Analysis
In this step, we have analysed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis
Static analysis of smart contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis
Manual analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analysed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption
In this step, we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit
Mythril, Slither, SmartCheck, Surya, Solhint.

Issue Categories

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below.

Risk-level	Description
High	A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.
Medium	The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.
Low	Low-level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.
Informational	These are severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Number of issues per severity

Type	High	Medium	Low	Informational
Open	0	0	0	0
Acknowledged	0	0	1	6
Closed	3	0	5	0

Functional Testing Results

The complete functional testing report has been attached below:
Vanilla test case

Some of the tests performed are mentioned below:

- should be able to initialize with the right values. PASS
- should be able to transfer the funds from the exempted fee wallet to non exempted. PASS
- should be able to transfer the funds from a non exempted fee wallet to another. PASS
- should be able to transfer the funds from the exempted wallet to another exempted wallet. PASS
- should be able to transfer the funds using the transferFrom. PASS
- should be able to get the right fund value from the balanceOf function. PASS
- should be able to get the right total supply of the token after burning funds. PASS
- should be able to accumulate the right amount of liquidity into the contract. PASS
- should be able to trigger swaps when funds are more than the required threshold in the contract. PASS
- should be able to accumulate the right amount of funds in the marketing wallet. PASS
- should be able to exclude and include the wallet from the reward. PASS
- should be able to disable/enable fees successfully. PASS
- Should be able to set all fee types by the owner. PASS
- Should be able to set other variables type by the owner. PASS

Issues Found

High severity issues

- **The Vanilla token is not compliant with BEP-20**

  As per the BEP-20 standard, The contract should implement the getOwner() function to support the movement of the token from the BSC to the Binance chain.

  Recommendation
  Consider implementing the getOwner() function within the contract to make it compliant with the BEP-20 standard, It will make the contract future proof.

  Status: Fixed

- **Liquidity funds would be sold at a discounted rate or even sold for free.**

  [#L418] in function swapTokensForEth() performs a swap to get WBNB over the pancake swap by giving 100 % slippage i.e amountOut = 0, This configuration opens a gate for a sandwich attack.

  1. An attacker could arbitrage the pancake liquidity pool and make an appreciation/depreciation in the prices.
  2. Then allow the current swap transaction to go through, which will give almost zero WBNB to the contract as the contract allows accepting 0 WBNB from the swap.
  3. Then the attacker can do another trade to reap the benefit by purchasing the assets from the discounted rate.

  Recommendation
  Use an external oracle like chainlink / TWAP oracle or some other to calculate the expected amountOut and put a trade on -5% (or any appropriate slippage percentage) over the pancake.

  Status: Fixed

- **Funds can remain locked for perpetuity.**

  [#L231] Contract is receiving funds externally but there is no function in the contract that will allow the contract owner or someone else to extract the funds from the contract. Although the receive() method is not required to interact with the router because the pair is created with the `WETH/WBNB` pair i.e. an ERC20 token. Due to this reason if anyone transfers the funds within the contract as it is allowed to receive it will remain locked for perpetuity.

  Recommendation
  If there is no other requirement of keeping receive() other than supporting the pancake swap then we recommend removing it from the contract. Otherwise, we recommend having an explicit recovery function that can recover WBNB/BNB from the contract.

  Status: Fixed

Medium severity issues

No issues found

Low severity issues

- **Unnecessary removes/restore operations lead to more gas consumption and complexity in code.**

  [#L443] removeAllFee() & at [#L465] restoreAllFee() is not needed when sender or receiver is excluded from the fee during the transfer as it is a total waste of the gas for removing and restoring the fee values.

  Recommendation
  Rewrite the _tokenTransfer() function.

```
function _tokenTransfer(address sender, address recipient, uint256 amount) private {
        bool excludedFromFee = _isExcludedFromFee[sender] ||
_isExcludedFromFee[recipient];

        if (!excludedFromFee) {
            require(amount <= _maxTxAmount, "Transfer amount exceeds the maxTxAmount.");
        }

        bool isSell = (recipient == uniswapV2Pair);
        ValuesStruct memory vs;

        if (excludedFromFee) {
            vs = ValuesStruct ({
                rAmount: amount * _getRate();
                rTransferAmount: amount * _getRate();
                rFee: 0;
                rBurn: 0;
                rMarketing: 0;
                rLiquidity: 0;
                tTransferAmount: amount;
                tFee: 0;
                tBurn: 0;
                tMarketing: 0;
                tLiquidity: 0;
            });
        } else {
            vs = _getValues(amount, isSell);
            _takeLiquidity(vs.rLiquidity, vs.tLiquidity);
            _distributeFee(vs.rFee, vs.rBurn, vs.rMarketing, vs.tFee, vs.tBurn, vs.tMarketing);
        }

        if (_isExcluded[sender] && !_isExcluded[recipient]) {
            _transferFromExcluded(sender, recipient, amount, vs);
        } else if (!_isExcluded[sender] && _isExcluded[recipient]) {
            _transferToExcluded(sender, recipient, vs);
        } else if (!_isExcluded[sender] && !_isExcluded[recipient]) {
            _transferStandard(sender, recipient, vs);
        } else if (_isExcluded[sender] && _isExcluded[recipient]) {
            _transferBothExcluded(sender, recipient, amount, vs);
        }

        if (_isExcludedFromFee[sender] || _isExcludedFromFee[recipient])
            restoreAllFee();
    }
```

Status: Fixed

- **High gas usage in the transfer of assets.**

  Every transfer leads to the calculation of the fee if the sender/ receiver is not excluded from fee deduction, Which requires access of the _taxFee, _liquidityFee, _marketFee, _burnFee state variables to calculate the right amount of fee deducted on a given trade that leads to 4 sloads (4 * 2100 = 8400).

  Recommendation
  We recommend storing all the fee types in one storage slot by declaring them as uint64, using this approach 3 sloads could be saved in every transfer of tokens i.e 3 * 2100 = 6300.

  Status: Acknowledged

- **Possibility of trade freely without any fee deduction.**

  The specific sequence of transactions would reset the previous fee values and because of this token holders would be able to transact freely.

  Possible scenario -

  - The owner calls the disableAllFees() to disable the fee, It will set the previous fee variable with the current fee values and assign 0 to current fee values [#L322 - #L330].

  - If the owner calls the disableAllFees() again then this time - there could be many scenarios where an owner may call this function again - then both current and previous fee reset to zero as current fees are already zero because of earlier call.

  - The owner calls to enableAllFees() by assuming that the previous fee (non zero value) would be assigned to the current fees and would be reused. But in reality, all fees would be zero and token holders can trade tokens without deduction of any fee.

  Although the owner could re-set them using the set fee functions. In the meantime, token holders can trade freely.

Recommendation
We recommend adding a require check in the disableAllFees() function to not be able to call it again if the fees are already disabled.

```
require(_taxFee != uint256(0) && _liquidityFee != uint256(0) && _marketFee != uint256(0) &&
_burnFee != uint256(0), "Not allowed to disable");
```

Status: Fixed

- **Inconsistency in events emissions.**

  There are some ownable functions that are emitting the events and some are not.
  Functions don't have events -

  - setTaxFee
  - setBurnFee
  - setLiquidityFee
  - setMarketFee
  - setNumTokensSellToAddToLiquidity
  - setMarketingWallet
  - setMaxTxPercent
  - excludeFromFee
  - includeInFee

  Recommendation
  We recommend having a consistent behavior of events within the codebase.

  Status: Fixed

- **Excessive gas was used during the calculation of the fee.**

  [#L267 - #L270] contract is calculating the different fees by using the respective functions i.e calculateTaxFee() , calculateBurnFee() etc. And every function has this (isSell ? _sellFeeMultiplier : SELLFEEDENOM) statement common so we can reduce the gas usage by just calculating this at once and changing the definition and declaration of the function.

Recommendation
Change the function definition and declaration depicted below.

```
function calculateTaxFee(uint256 _amount, bool isSell) private view returns (uint256) {
        return _amount * _taxFee * (isSell ? _sellFeeMultiplier : SELLFEEDENOM) /
DENOMINATOR / SELLFEEDENOM;
```

changes to

```
function calculateTaxFee(uint256 _amount, uint256 _multiplier) private view returns
(uint256) {
        return _amount * _taxFee * _multiplier / 10 ** 6;
}

Where —-->
uint256 _multiplier = (isSell ? _sellFeeMultiplier : SELLFEEDENOM);
```

Above optimization saves 3 sloads which is a significant amount of gas during the transfer of the assets.

Status: Acknowledged

- **A Nonmutable variable should be declared as constant.**

  [#L76] SELLFEEDENOM should be declared as constant. In the current code, it is a variable and takes a sload operation to read the value that is too costly in comparison to free read if it declares as the constant.

  Recommendation
  Declare SELLFEEDENOM as a constant type.

  Status: Fixed

Informational issues

- **The owner can front-run the transactions to charge more fees.**

  The owner of the contract can front-run the transaction and can charge more fees as it should be during the submission of the transaction.

Ex - Alice submitted a transaction of transferring 1000 tokens to Bob and at the same time the owner could submit a transaction to change various different types of fees and tip that transaction more so it gets included first in comparison to the transfer transaction because of this Alice ends up paying more fee.

Recommendation
This is a known issue within almost every ownership-based model, Generally we recommend having a time-based lock mechanism where the change in the variable will get in effect after a certain interval of time. Although if the owner does this then it will counter-effect the price of the token as a token will lose its credibility.

Status: Acknowledged

- **Ambiguity in naming convention.**

  [#L100] BSC chain doesn't have the uniswap protocol deployed so it creates confusion during the readability of the contract, I think currently the contract is using the Pancake Router so it is better to rename the interface to IPancakeRouter instead of IUniswapRouterV2. similar applies to IUniswapV2Factory, should be renamed to IPancakeFactory.

  Status: Acknowledged

- **Inconsistency in using router address.**

  [#L209] Using a hardcoded address of the pancake router, While that is the incorrect address for the testnet.

  Recommendation
  Rewrite the require statement like below.

```
require(account != address(uniswapV2Router), 'We can not exclude Pancake router.');
```

To have consistency in the code and avoid chances of error as in current testnet deployment, the deployer missed replacing the mainnet pancake router with the testnet router.

Status: Acknowledged

- **Multiple pragma directives have been used.**

  Recommendation
  Contracts should be deployed using the same compiler version/flags with which they have been tested. Locking the pragma (e.g. by not using ^ in pragma solidity 0.8.0) ensures that contracts do not accidentally get deployed using an older compiler version with unfixed bugs. Ref: Security Pitfall 2

  Status: Acknowledged

- **Unnecessary code complexity.**

  [#L57] using ~uint256(0) to calculate the maximum value of uint256 type variable supports instead of using the method provided by the solidity to get the maximum range of the uint256.

  Recommendation
  Use type(uint256).max to get the maximum value of the uint256.

  Status: Acknowledged

- **Unnecessary complexity in struct declaration.**

  [#L31] ValuesStruct has written with the combination of the variables of RValuesStruct and TValuesStruct that is not the cleaner approach.

  Recommendation
  ValuesStruct can be rewritten as the combination of the other two structs.

```
ValuesStruct {
    RValuesStruct rValue;
    TValuesStruct tValue;
 }
```

  This is a cleaner and more readable approach instead of re-declaring the same variables.

  Status: Acknowledged

Closing Summary

Some issues of High & Low severity were found,some of the issues are fixed. Some suggestions and best practices are also provided in order to improve the code quality and security posture.

Disclaimer

Quillhash audit is not a security warranty, investment advice, or an endorsement of the Vanilla Bet. This audit does not provide a security or correctness guarantee of the audited smart contracts. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the Vanilla Bet team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

Audit Report
February, 2022

For

VANILLA BET

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com