QuillAudits

Audit Report
April, 2022

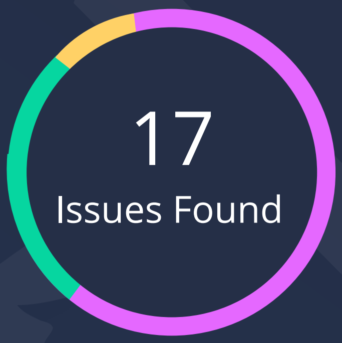
For

Severus™

Table of Content

Executive Summary

Project Name	SeverusDAO
Overview	The Severus Dao Token is a utility, rewards-based, deflationary token with an ever-increasing floor price. Severus was founded in South Africa to help accelerate global crypto adoption by providing a secure, integrated blockchain experience with the same level of security as a AAA bank account coupled with the intelligence and innovation of a bleeding edge Fintech.
Timeline	April 19, 2022 - April 26, 2022
Method	Manual Review, Functional Testing, Automated Testing etc.
Scope of Audit	The scope of this audit was to analyse Severus.finance contracts codebase for quality, security, and correctness.
Deployed at	*https://bscscan.com/address/0xF0DC35a6868A7b0c0213dDEbb7a6f941664b8A36*



🟥 High 🟨 Medium
🟩 Low 🟪 Informational

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	1	5	11
Partially Resolved Issues	0	0	0	0
Resolved Issues	0	0	0	0

Types of Severities

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.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Checked Vulnerabilities

- ✓ Re-entrancy
- ✓ Timestamp Dependence
- ✓ Gas Limit and Loops
- ✓ Exception Disorder
- ✓ Gasless Send
- ✓ Use of tx.origin
- ✓ Compiler version not fixed
- ✓ Address hardcoded
- ✓ Divide before multiply
- ✓ Integer overflow/underflow
- ✓ 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

Severus.finance - Audit Report

Techniques and Methods

Throughout the audit of 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 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

Remix IDE, Truffle, Truffle Team, Solhint, Mythril, Slither, Solidity statistic analysis.

Manual Testing

High Severity Issues

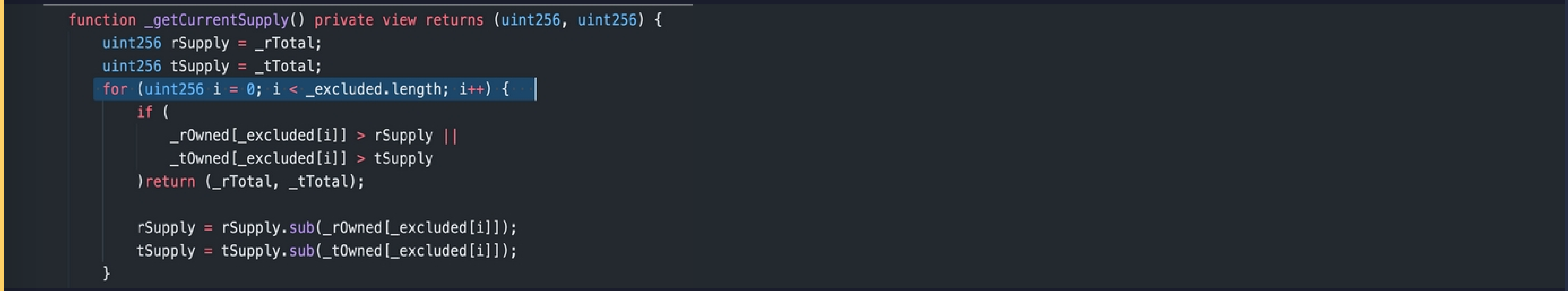
No issues found

Medium Severity Issues

1. Excluded[] Length problem: Transaction may fail if the length of Excluded array is very large.

Description

Severus contract has two functions that use a for loop over an array. includeInReward() and _getCurrentSupply(). If the length of the excluded array is very large, This may lead to extreme gas costs up to the block gas limit and eventually fail the transaction.
In an extreme situation with a large number of excluded addresses, transaction gas may exceed the maximum block gas size and all transfers will be effectively blocked. If the owner's account gets compromised the attacker can make the token completely unusable for all users.



```
function _getCurrentSupply() private view returns (uint256, uint256) {
    uint256 rSupply = _rTotal;
    uint256 tSupply = _tTotal;
    for (uint256 i = 0; i < _excluded.length; i++) {
        if (
            _rOwned[_excluded[i]] > rSupply ||
            _tOwned[_excluded[i]] > tSupply
        )return (_rTotal, _tTotal);

        rSupply = rSupply.sub(_rOwned[_excluded[i]]);
        tSupply = tSupply.sub(_tOwned[_excluded[i]]);
    }
```

Exploit scenario

Let's assume a scenario where the number of addresses excluded is very large and by running a for loop over that length, the transaction fails due to an out of gas issue. Now a user wants to transfer his token to another address. The user will call the transfer function, now if we look at the whole flow, eventually any one of the four transfer methods will be called and all those transfer functions will call _getValues(). And _getValues() will call _getCurrentSupply() which has a for loop which consumes a large quantity of gas. So if _getCurrentSupply() fails every transfer will fail.

Remediation

This issue can be avoided if the owner restricts the number of addresses excluded from the rewards. It is recommended to implement checks that prevent owner from adding addresses to exclude from rewards to a certain threshold.

Status

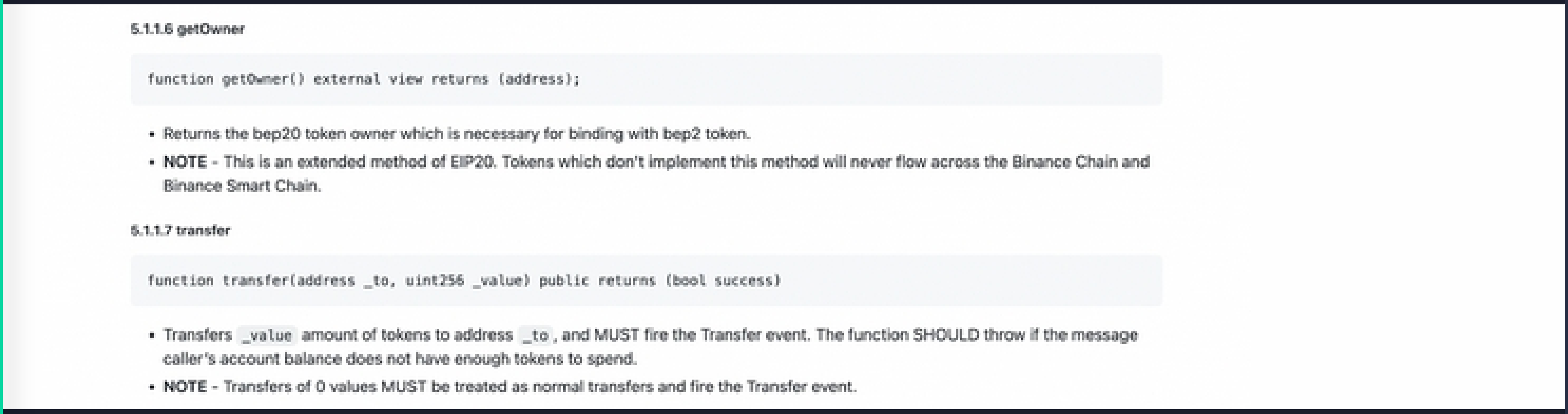
Acknowledged

Low Severity Issues

2. BEP20 Standard violation

Description

Implementation of transfer() function does not allow the input of zero amount as it's demanded in ERC20 and BEP20 standards. This issue may break the interaction with smart contracts that rely on full BEP20 support. Moreover, the GetOwner() function which is a mandatory function is missing from the contract.



5.1.1.6 getOwner

```
function getOwner() external view returns (address);
```

- Returns the bep20 token owner which is necessary for binding with bep2 token.
- **NOTE** - This is an extended method of EIP20. Tokens which don't implement this method will never flow across the Binance Chain and Binance Smart Chain.

5.1.1.7 transfer

```
function transfer(address _to, uint256 _value) public returns (bool success)
```

- Transfers `_value` amount of tokens to address `_to`, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend.
- **NOTE** - Transfers of 0 values MUST be treated as normal transfers and fire the Transfer event.

Recommendation

The transfer function must treat zero amount transfer as a normal transfer and an event must be emitted. Moreover, it is recommended to implement getOwner() function.
Reference: *https://github.com/bnb-chain/BEPs/blob/master/BEP20.md#5117-transfer*

Status

Acknowledged

3. Incorrect Error Message

Description

includeInReward() function displays an incorrect error message if the require statement fails.

Remediation

The message "Account is already excluded" can be changed to "Account is not excluded" or "Account is already included".

Status

Acknowledged

4. Contract gains non-withdrawable BNB via the swapAndLiquify function

Description

The swapAndLiquify function converts half of the contractTokenBalance SVD tokens to BNB. The other half of SVD tokens and part of the converted BNB are deposited into the SVD BNB pool as liquidity. For every swapAndLiquify function call, a small amount of BNB leftover in the contract. This is because the price of SVD drops after swapping the first half of SVD tokens into BNBs, and the other half of SVD tokens require less than the converted BNB to be paired with it when adding liquidity. The contract doesn't appear to provide a way to withdraw those BNB, and they will be locked in the contract forever. Moreover, The payable receive() function in L1271 makes it possible for the contract to receive bnb. All these will be accumulated over time and there is no way to withdraw it.

Remediation

Consider adding a withdraw function to rescue stuck BNB. Alternatively, it can be distributed among token holders in proportion to their holdings.

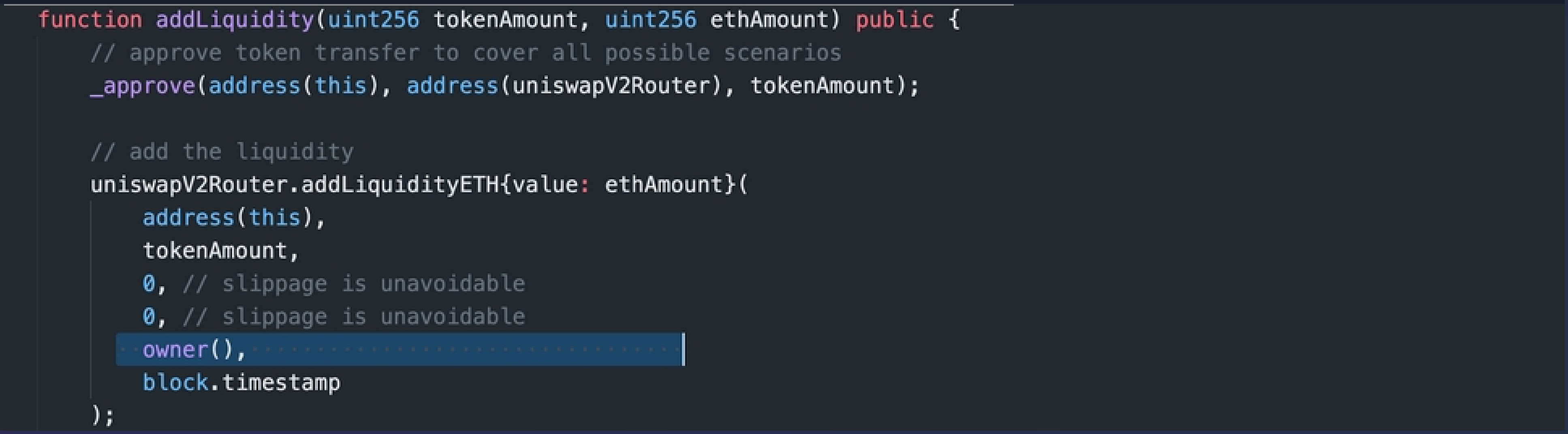
Status

Acknowledged

5. addLiquidity() recipient issues.

Description

addLiquidity() function in LiquidityGeneratorToken contract calls for uniswapV2Router.addLiquidityETH() function with the parameter of lp tokens recipient set to owner address. With time the owner address may accumulate a significant amount of LP tokens which may be dangerous for token economics if an owner acts maliciously or its account gets compromised. This issue can be fixed by changing the recipient address to the Severus contract or by renouncing ownership which will effectively lock the generated LP tokens.



```
function addLiquidity(uint256 tokenAmount, uint256 ethAmount) public {
    // approve token transfer to cover all possible scenarios
    _approve(address(this), address(uniswapV2Router), tokenAmount);

    // add the liquidity
    uniswapV2Router.addLiquidityETH{value: ethAmount}(
        address(this),
        tokenAmount,
        0, // slippage is unavoidable
        0, // slippage is unavoidable
        owner(),
        block.timestamp
    );
```

Remediation

Renounce the ownership of the contract or replace the recipient address to the address of the contract.

Status

Acknowledged

6. The owner can abuse ExcludeFromReward() functionality and prevent users from earning rewards.

Description

As per the flow of the contract, whenever a token transfer is made, all the token holders receive a share of the transaction fees. The owner of the token contract can redistribute part of the tokens from users to a specific account. For this owner can exclude an account from the reward and include it back later. This will redistribute part of the tokens from holders in profit of the included account and the excluded account will not receive the reward.

Exploit Scenario

The Owner can temporarily exclude an account from earning a reward and include it back later after some time.

Remediation

We suggest lock exclusion/inclusion methods by locking ownership for the maximum possible amount of time. Moreover, excluding/including an account from earning rewards is a very sensitive functionality that involves the interference of certain privileged users (owner). Whenever any significant action is performed by the privileged users, an event must be emitted.

Status

Acknowledged

Informational Issues

7. Missing Testcases

Description

Test cases for the code and functions have not been provided.

Recommendation

It is recommended to write testcases of all the functions. Any existing tests that fail must be resolved. Tests will help in determining if the code is working in the expected way. Unit tests, functional tests, and integration tests should have been performed to achieve good test coverage across the entire codebase.

Status

Acknowledged

8. Public functions that could be declared external inorder to save gas

Whenever a function is not called internally, it is recommended to define them as external instead of public in order to save gas. For all the public functions, the input parameters are copied to memory automatically, and it costs gas. If your function is only called externally, then you should explicitly mark it as external. External function's parameters are not copied into memory but are read from calldata directly. This small optimization in your solidity code can save you a lot of gas when the function input parameters are huge.
Following functions could be declared external:

- totalSupply()
- name()
- symbol()
- decimals()
- increaseAllowance()
- decreaseAllowance()
- excludeFromReward()
- totalFees()
- deliver()
- excludeFromFee()
- includeInFee()
- isExcludedFromReward()

Status

Acknowledged

9. Variables That Could Be Declared As Constant inorder to save gas

There are multiple variables in the contract, whose value is never updated, it is recommended to declare those variables as constant.
Here is a list of variable that could be declared constant:

- Name
- Symbol
- Decimals
- _tTotal

Status

Acknowledged

10. Missing zero check

Description

Contracts lack zero address checks, hence are prone to be initialized with zero addresses.

- constructor()
- excludeFromFee()
- includeInFee()

Severus.finance - Audit Report

Recommendation

Consider adding zero address checks in order to avoid risks of incorrect contract initializations.

Status

Acknowledged

11. Unindexed event parameters

Description

MinTokensBeforeSwapUpdated, SwapAndLiquify and SwapAndLiquifyEnabledUpdated events don't have any indexed parameters. Unindexed parameters make it difficult to track important data for off-chain monitoring tools. Moreover, in SwapAndLiquify there is a typo in 3rd parameter tokensIntoLiqudity, it should be tokensIntoLiqiuidity.

```
event MinTokensBeforeSwapUpdated(uint256 minTokensBeforeSwap);
event SwapAndLiquifyEnabledUpdated(bool enabled);
event SwapAndLiquify(
    uint256 tokensSwapped,
    uint256 ethReceived,
    uint256 tokensIntoLiqudity
);
```

Recommendation

Consider indexing event parameters to avoid the task of off-chain services searching and filtering for specific events.

Status

Acknowledged

12. Third Party Dependencies

Description

The logic of the contract requires it to interact with third-party protocols. The scope of the audit treats 3rd party entities as black boxes and assumes their functional correctness. However, in the real world, 3rd parties can be compromised and this may lead to lost or stolen assets. In addition, upgrades of 3rd parties can possibly create severe impacts, such as increasing fees of 3rd parties, migrating to new LP pools, etc.

Recommendation

We understand that the business logic of the LiquidityGeneratorToken contract requires interaction with third-party protocols. We encourage the team to constantly monitor the statuses of 3rd parties to mitigate the side effects when unexpected activities are observed.

Status

Acknowledged

13. Unnecessary require statement

Description

In the constructor, on line 988,989,990 there are three require statements which checks if provided value is greater than or equal to 0. However, those values are of type uint and uint values are always >= 0. Hence those require statements can be removed.

Recommendation

Consider removing this extra code from the contract.

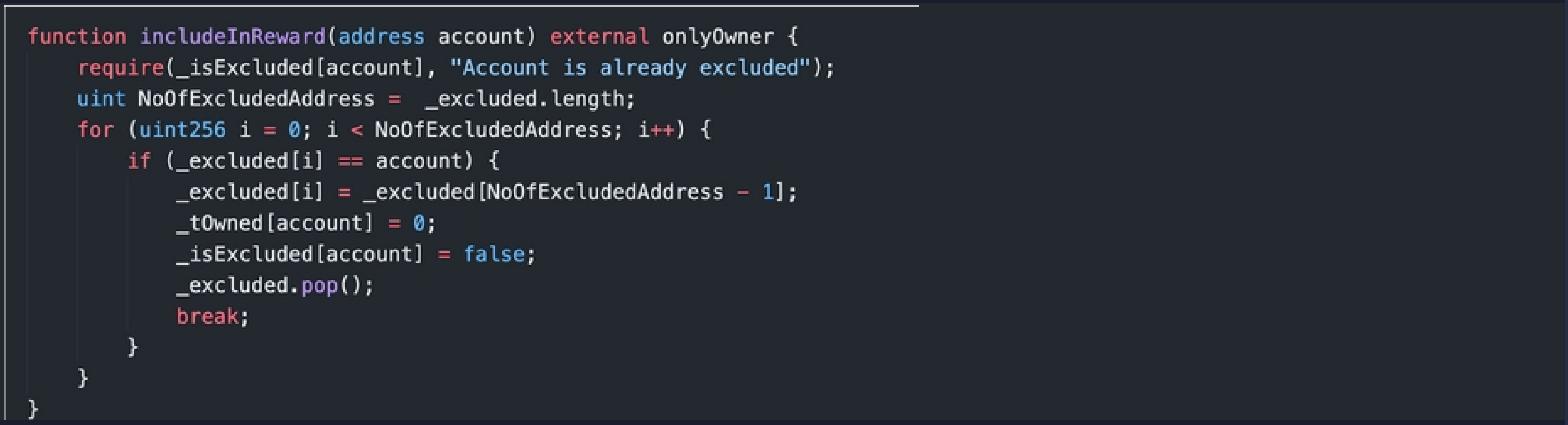
Status

Acknowledged

14. Gas optimization: For loop optimization

Description

In includeInReward() and _getCurrentSupply(), there is a for loop which iterates the value of _excluded.length times. Each time the for loop executes _excluded.length is calculated which consumes some gas. This can be optimized by calculating the value of _excluded.length outside the for loop. The optimized loop would look like:



```
function includeInReward(address account) external onlyOwner {
    require(_isExcluded[account], "Account is already excluded");
    uint NoOfExcludedAddress =  _excluded.length;
    for (uint256 i = 0; i < NoOfExcludedAddress; i++) {
        if (_excluded[i] == account) {
            _excluded[i] = _excluded[NoOfExcludedAddress - 1];
            _tOwned[account] = 0;
            _isExcluded[account] = false;
            _excluded.pop();
            break;
        }
    }
}
```

Recommendation

Update the loop as recommended above.

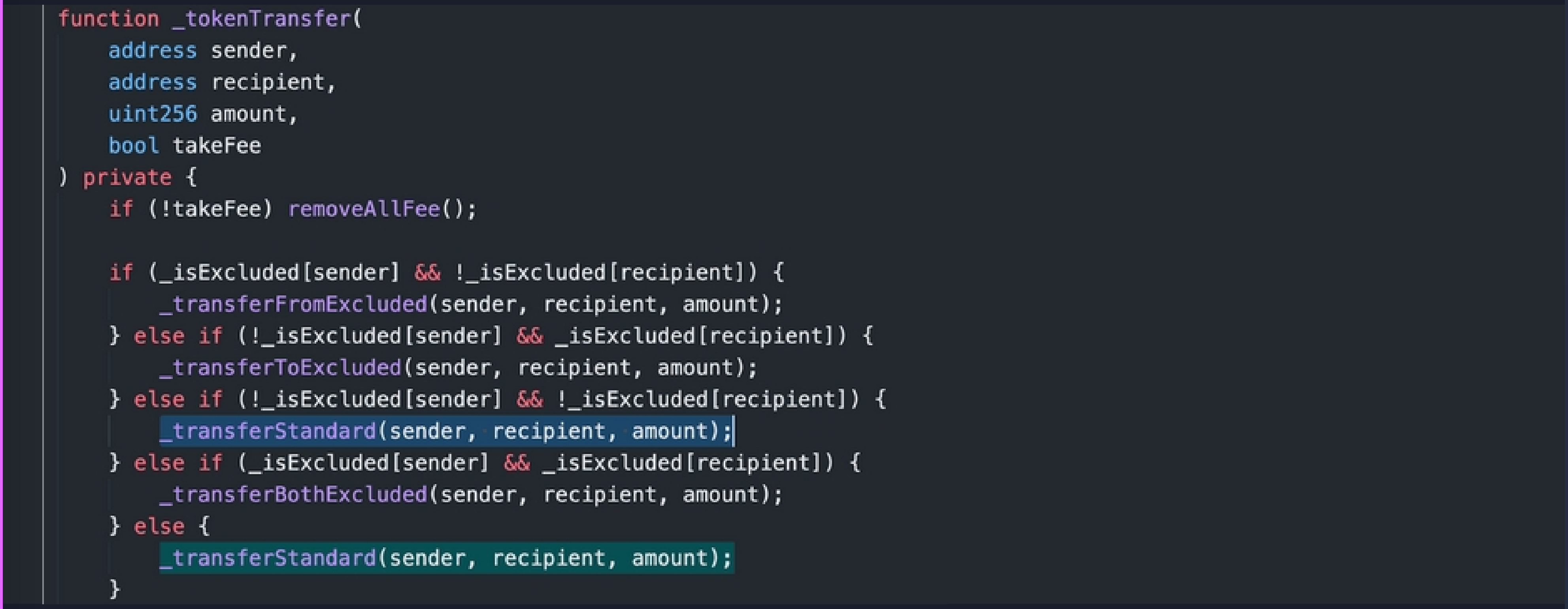
Status

Acknowledged

15. Redundant Code

Description

The token _tokenTransfer() function in LiquidityGeneratorToken.sol, has an if-else block and in the second else if block, there is _transferStandard(sender, recipient, amount). This condition is already satisfied by the else block hence it is safe to remove the extra code from the contract



```
function _tokenTransfer(
    address sender,
    address recipient,
    uint256 amount,
    bool takeFee
) private {
    if (!takeFee) removeAllFee();

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

Recommendation

Remove the Redundant code from the contract.

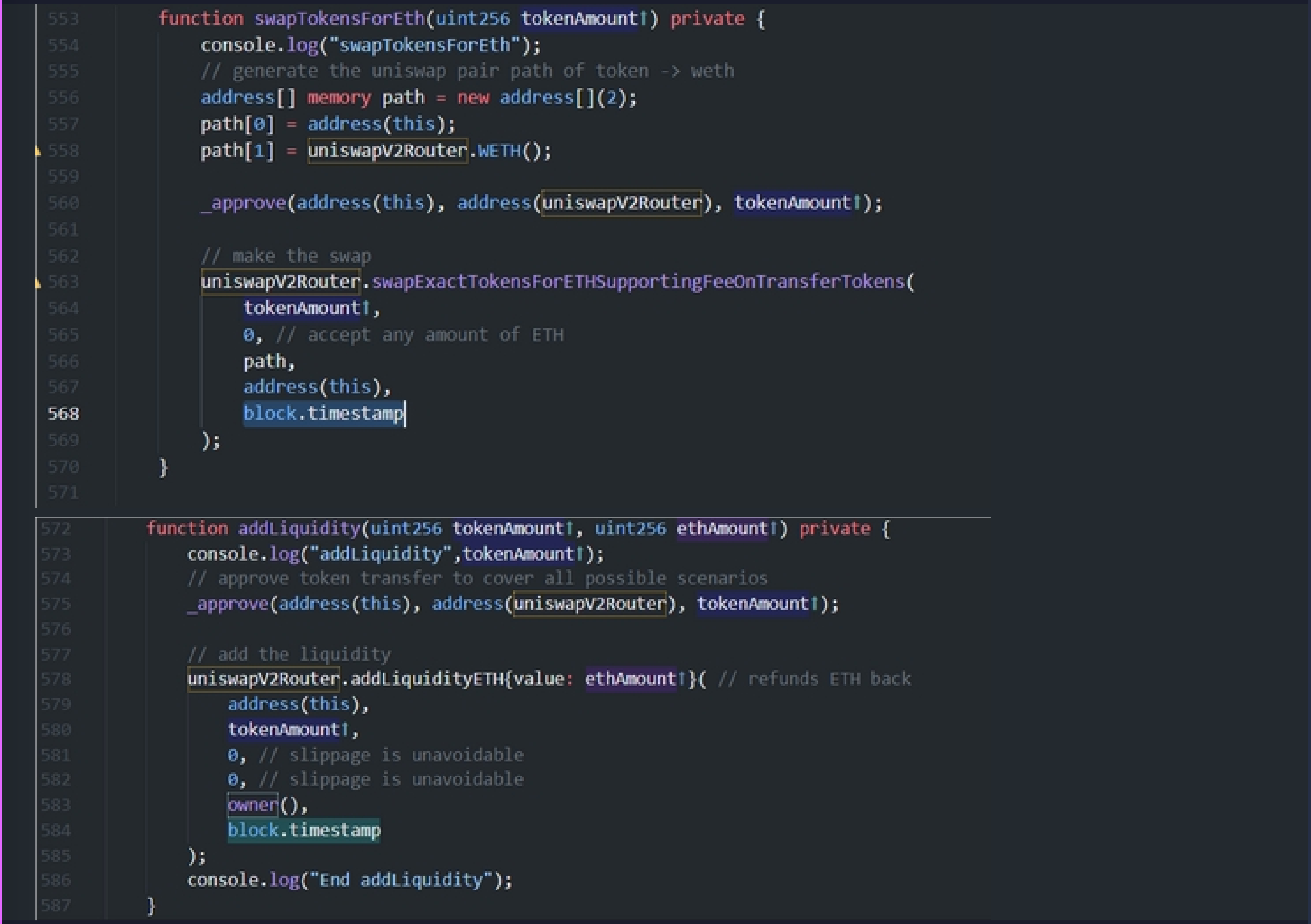
Status

Acknowledged

16. Use of block.timestamp for trade deadline

Description

Deadline set as block.timestamp, In this case of delay too less time margin for the transaction to execute, may add the risk of the transaction being reverted by router contract because of the expired deadline. Too big deadline/time margin may add risk of miner manipulation, where a miner can hold transaction and can execute or add it to block at profitable time.



```
    function swapTokensForEth(uint256 tokenAmount) private {
        console.log("swapTokensForEth");
        // generate the uniswap pair path of token -> weth
        address[] memory path = new address[](2);
        path[0] = address(this);
        path[1] = uniswapV2Router.WETH();

        _approve(address(this), address(uniswapV2Router), tokenAmount);

        // make the swap
        uniswapV2Router.swapExactTokensForETHSupportingFeeOnTransferTokens(
            tokenAmount,
            0, // accept any amount of ETH
            path,
            address(this),
            block.timestamp
        );
    }

    function addLiquidity(uint256 tokenAmount, uint256 ethAmount) private {
        console.log("addLiquidity",tokenAmount);
        // approve token transfer to cover all possible scenarios
        _approve(address(this), address(uniswapV2Router), tokenAmount);

        // add the liquidity
        uniswapV2Router.addLiquidityETH{value: ethAmount}( // refunds ETH back
            address(this),
            tokenAmount,
            0, // slippage is unavoidable
            0, // slippage is unavoidable
            owner(),
            block.timestamp
        );
        console.log("End addLiquidity");
    }
```

Recommendation

Consider adding block.timestamp + some amount of seconds while adding a deadline.

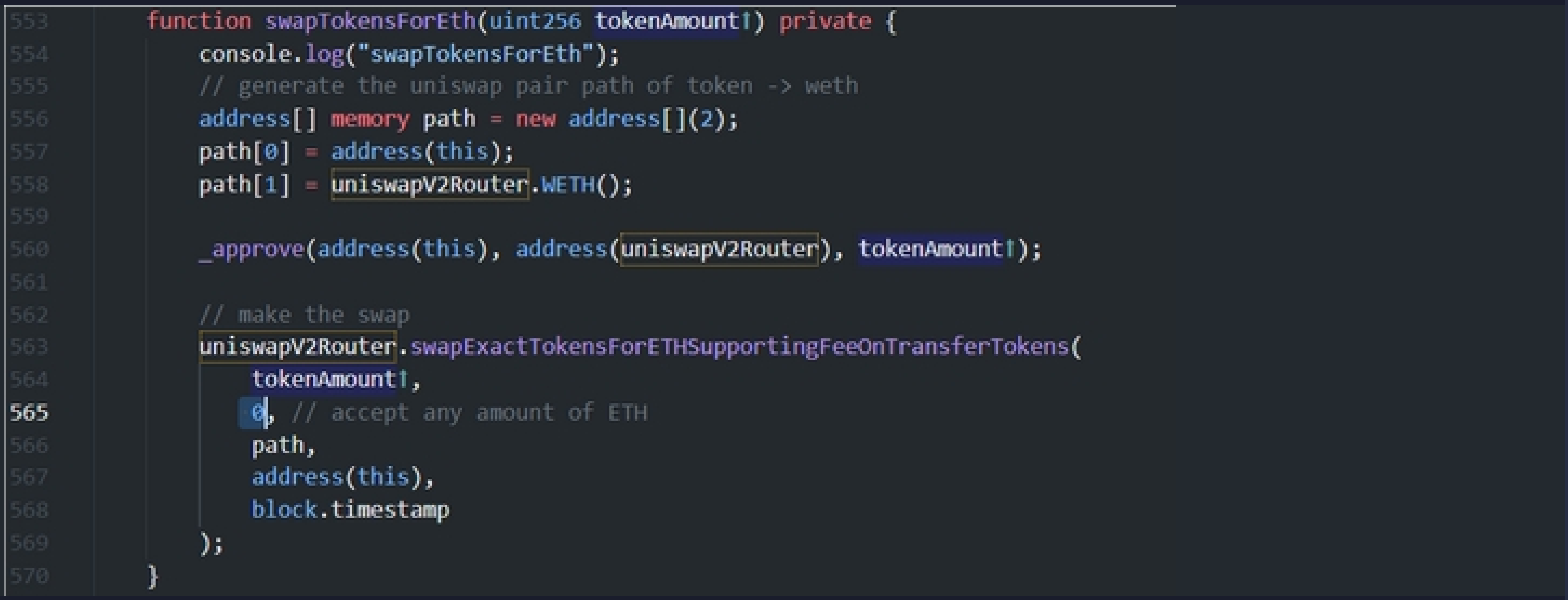
Status

Acknowledged

17. Minimum amount to receive is 0

Description

Minimum amount of output tokens that must be received is 0, which allows trade to execute even when the output amount is very low.



```
    function swapTokensForEth(uint256 tokenAmount) private {
        console.log("swapTokensForEth");
        // generate the uniswap pair path of token -> weth
        address[] memory path = new address[](2);
        path[0] = address(this);
        path[1] = uniswapV2Router.WETH();

        _approve(address(this), address(uniswapV2Router), tokenAmount);

        // make the swap
        uniswapV2Router.swapExactTokensForETHSupportingFeeOnTransferTokens(
            tokenAmount,
            0, // accept any amount of ETH
            path,
            address(this),
            block.timestamp
        );
    }
```

Recommendation

Consider adding a minimum amount to receive greater than zero. The minimum amount to receive may vary according to The token amount passed in while swapping for ETH.

Status

Acknowledged

Functional Testing

LiquidityGeneratorToken.sol

- ✓ Should test all the getter values.
- ✓ Should test approve, allowance and transferFrom.
- ✓ Total Fees deducted must not be >25%.
- ✓ Should test increaseallownace and decrease allowance.
- ✓ Should correctly calculate reflections from tokens and vice versa.
- ✓ Should test transferStandard, transferFromexcluded, transferToExcluded, transferBothExcluded.
- ✓ Should check charity fees.
- ✓ Should check liquidity fees.
- ✓ Should check transaction fees.
- ✓ Fees must not be charged from excluded address.
- ✓ Should update balance correctly.
- ✓ Excluded from rewards must not receive rewards.
- ✓ OnlyOwner modifier must work as intended and must revert if called by non owner addresses.

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

Several issues of Medium, Low and Informational severity were found. Some suggestions and best practices are also provided in order to improve the code quality and security posture.The Severus finance team has acknowledged all of the issues.

Disclaimer

QuillAudits smart contract audit is not a security warranty, investment advice, or an endorsement of the Severus.finance platform. 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 Severus.finance Team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem.

500+
Audits Completed

$15B
Secured

500K
Lines of Code Audited

Follow Our Journey

Audit Report
April, 2022

For

Severus™

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com