Deliverable: Smart Contract Audit Report

Only Up

Smart Contract Review

Security Report

June 2021

Disclaimer

Report Summary

Title	**Only Up Smart Contract Audit**		
Project Owner	Only Up		
Type	Public		
Reviewed by	Vatsal Raychura	**Revision date**	01/06/2021
Approved by	eNebula Solutions Private Limited	**Approval date**	01/06/2021
		Nº Pages	**23**

Overview

Background

The ‘Only Up’ requested that eNebula Solutions perform an Extensive Smart Contract audit of their ‘SafeMoon’ Contract.

Project Dates

The following is the project schedule for this review and report:

- **June 01**: Smart Contract Review Completed *(Completed)*
- **June 01**: Delivery of Smart Contract Audit Report *(Completed)*

Review Team

The following eNebula Solutions team member participated in this review:

- Sejal Barad, Security Researcher and Engineer
- Vatsal Raychura, Security Researcher and Engineer

Coverage

Target Specification and Revision

For this audit, we performed research, investigation, and review of the smart contract of Only Up.

The following bscscan link were considered in-scope for the review:

- Only Up Project: https://bscscan.com/address/0x07340F61633Fc21b59Edef1D4FB5C36cf1F098e2#code

Introduction

Given the opportunity to review the Only Up Contracts related smart contract source code, we in the report outline our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts is ready to launch after resolving the **1 medium and 9 low severity issues**, there were no critical or high severity issues found related to business logic, security or performance.

About Only Up: -

Item	Bug Common Description
Issuer	Only Up
Website	https://oupdefi.com/
Type	ERC20
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 01, 2021

The Full List of Check Items:

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization

Advanced DeFi Scrutiny	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

Common Weakness Enumeration (CWE) Classifications Used in This Audit:

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not securitysoftware.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values,Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code,or if the application does not handle all possible return/statuscodes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.

Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. Theymay not directly introduce a vulnerability, but indicate theproduct has not been carefully developed or maintained.

Findings

Summary

Here is a summary of our findings after analyzing the SafeMoon Smart Contract Review. During the first phase of our audit, we studied the smart contract source code and ran our in-house static code analyzer through the Specific tool. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	No. of Issues
Critical	0
High	0
Medium	1
Low	9
Total	10

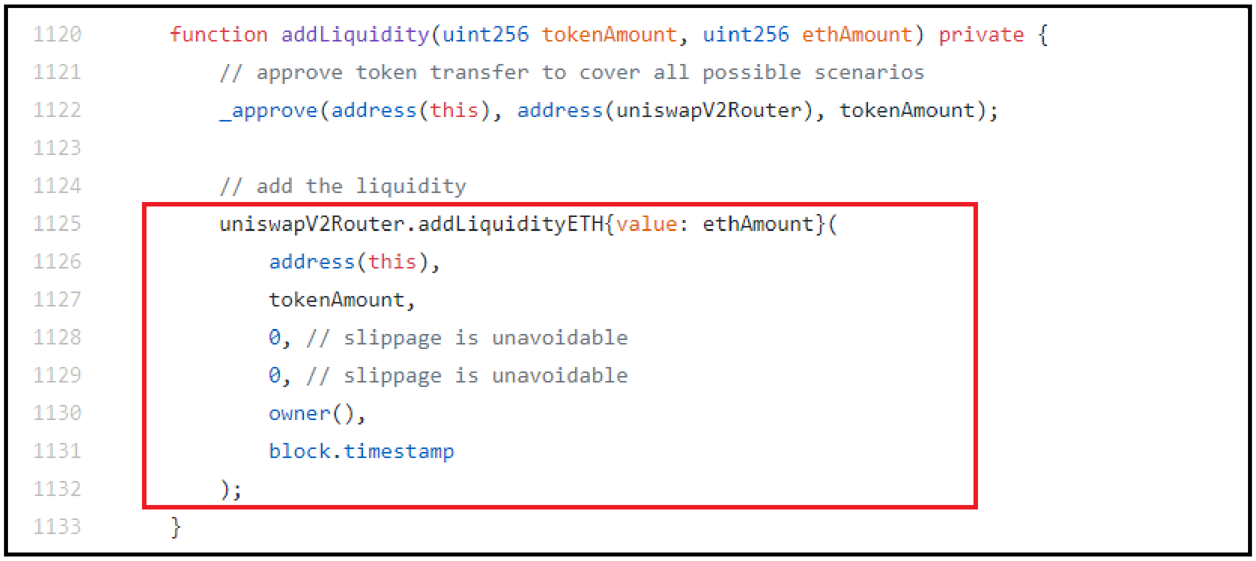
We have so far identified that there are potential issues with severity of **0 Critical, 0 High, 1 Medium, and 9 Low.** Overall, these smart contracts are well-designed and engineered, though the implementation can be improved and bug free by common recommendations given under the POCs.

Detailed Results

Basic Coding Bugs

1. **Unused return**

    - Severity: Medium
    - Result: Found
    - Confidence: Medium
    - Bug Common Description: The return value of an external call is not stored in a local or state variable.
    - POC:



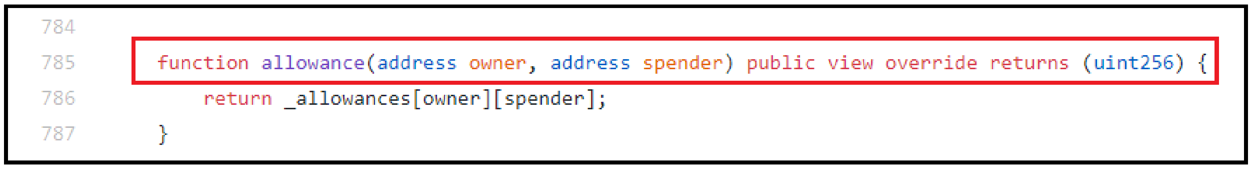
```
    function addLiquidity(uint256 tokenAmount, uint256 ethAmount) private {
        // approve token transfer to cover all possible scenarios
        _approve(address(this), address(uniswapV2Router), tokenAmount);

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

    - Recommendation: Ensure that all the return values of the function calls are used.

2. Local variable shadowing

- Severity: Low
- Result: Found
- Confidence: High
- Bug Common Description: Detection of shadowing using local variables.
- POC:

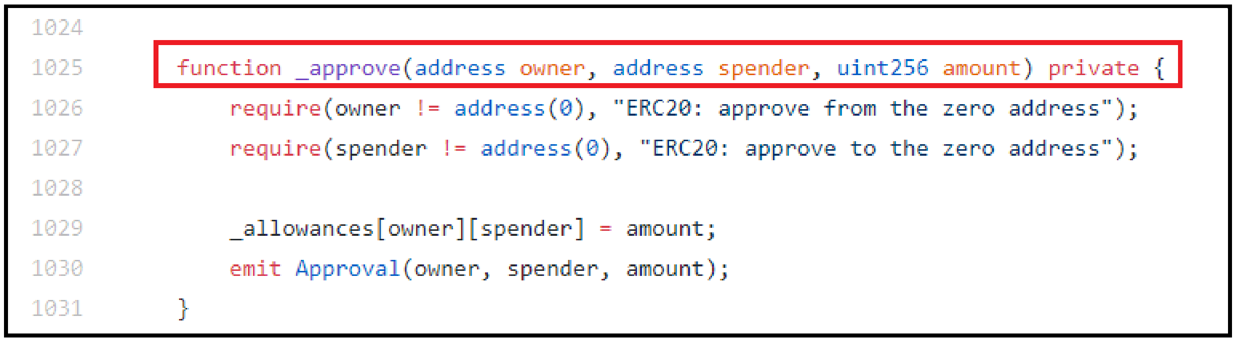


```
    function allowance(address owner, address spender) public view override returns (uint256) {
        return _allowances[owner][spender];
    }
```

- Recommendation: Rename the local variables that shadow another component.

3. **Local variable shadowing**

- Severity: Low
- Result: Found
- Confidence: High
- Bug Common Description: Detection of shadowing using local variables.
- POC:



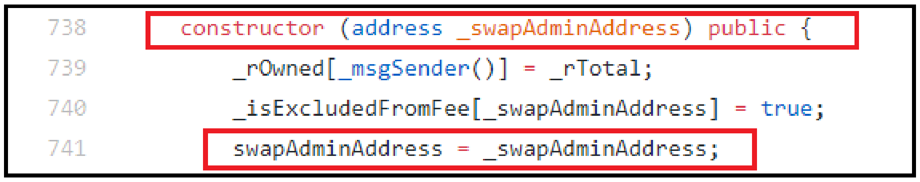
```
    function _approve(address owner, address spender, uint256 amount) private {
        require(owner != address(0), "ERC20: approve from the zero address");
        require(spender != address(0), "ERC20: approve to the zero address");

        _allowances[owner][spender] = amount;
        emit Approval(owner, spender, amount);
    }
```

- Recommendation: Rename the local variables that shadow another component.

4. Missing zero address validation

- Severity: Low
- Result: Found
- Confidence: Medium
- Bug Common Description: Detect missing zero address validation.
- POC:

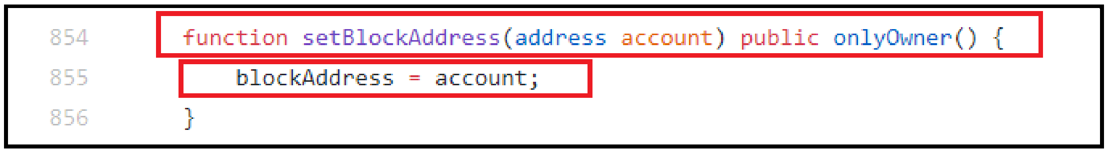


```
constructor (address _swapAdminAddress) public {
        _rOwned[_msgSender()] = _rTotal;
        _isExcludedFromFee[_swapAdminAddress] = true;
        swapAdminAddress = _swapAdminAddress;
```

- Recommendation: Check that the address is not zero.

5. Missing zero address validation

- Severity: Low
- Result: Found
- Confidence: Medium
- Bug Common Description: Detect missing zero address validation.
- POC:



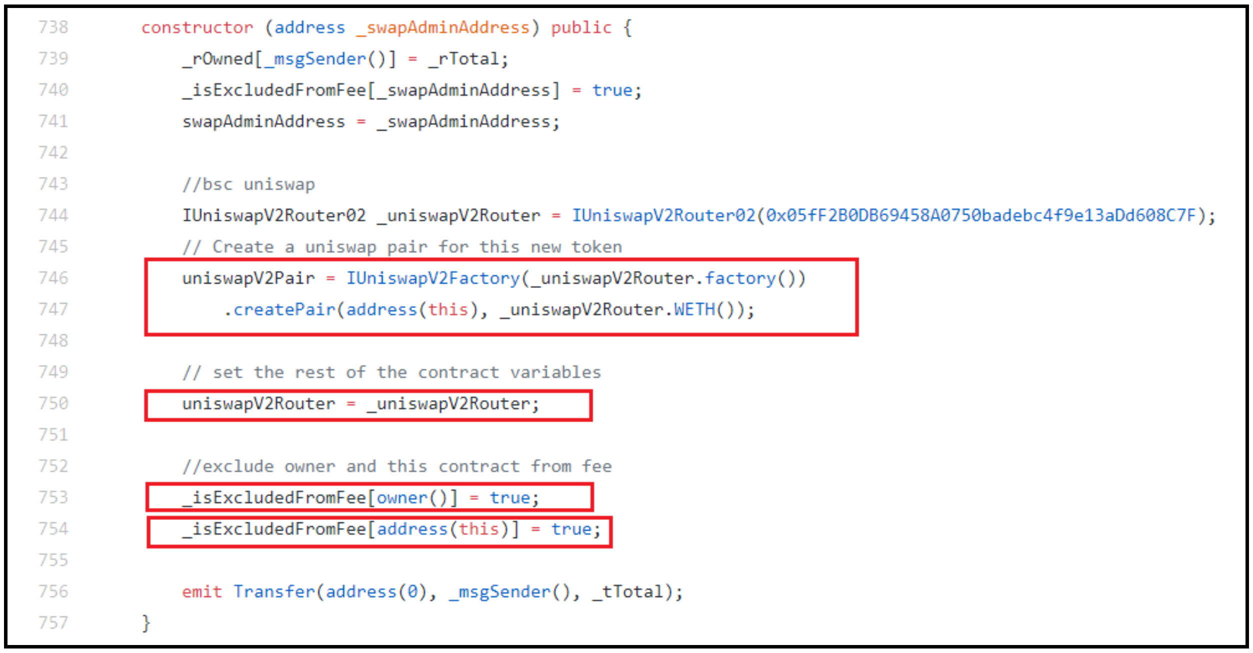
```
function setBlockAddress(address account) public onlyOwner() {
    blockAddress = account;
}
```

- Recommendation: Check that the address is not zero.

6. Reentrancy vulnerabilities

- Severity: Low
- Result: Found
- Confidence: Medium
- Bug Common Description: Detection of the reentrancy bug. Only report reentrancy that acts as a double call (see reentrancy-eth, reentrancy-no-eth).
- POC:
  External calls:
  - uniswapV2Pair = IUniswapV2Factory(_uniswapV2Router.factory()).createPair(address(this),_uniswapV2Router.WETH()) (#746-747)

  State variables written after the call(s):
  - _isExcludedFromFee[owner()] = true (#753)
  - _isExcludedFromFee[address(this)] = true (#754)
  - uniswapV2Router = _uniswapV2Router (#750)



```
738    constructor (address _swapAdminAddress) public {
739        _rOwned[_msgSender()] = _rTotal;
740        _isExcludedFromFee[_swapAdminAddress] = true;
741        swapAdminAddress = _swapAdminAddress;
742
743        //bsc uniswap
744        IUniswapV2Router02 _uniswapV2Router = IUniswapV2Router02(0x05fF2B0DB69458A0750badebc4f9e13aDd608C7F);
745        // Create a uniswap pair for this new token
746        uniswapV2Pair = IUniswapV2Factory(_uniswapV2Router.factory())
747            .createPair(address(this), _uniswapV2Router.WETH());
748
749        // set the rest of the contract variables
750        uniswapV2Router = _uniswapV2Router;
751
752        //exclude owner and this contract from fee
753        _isExcludedFromFee[owner()] = true;
754        _isExcludedFromFee[address(this)] = true;
755
756        emit Transfer(address(0), _msgSender(), _tTotal);
757    }
```

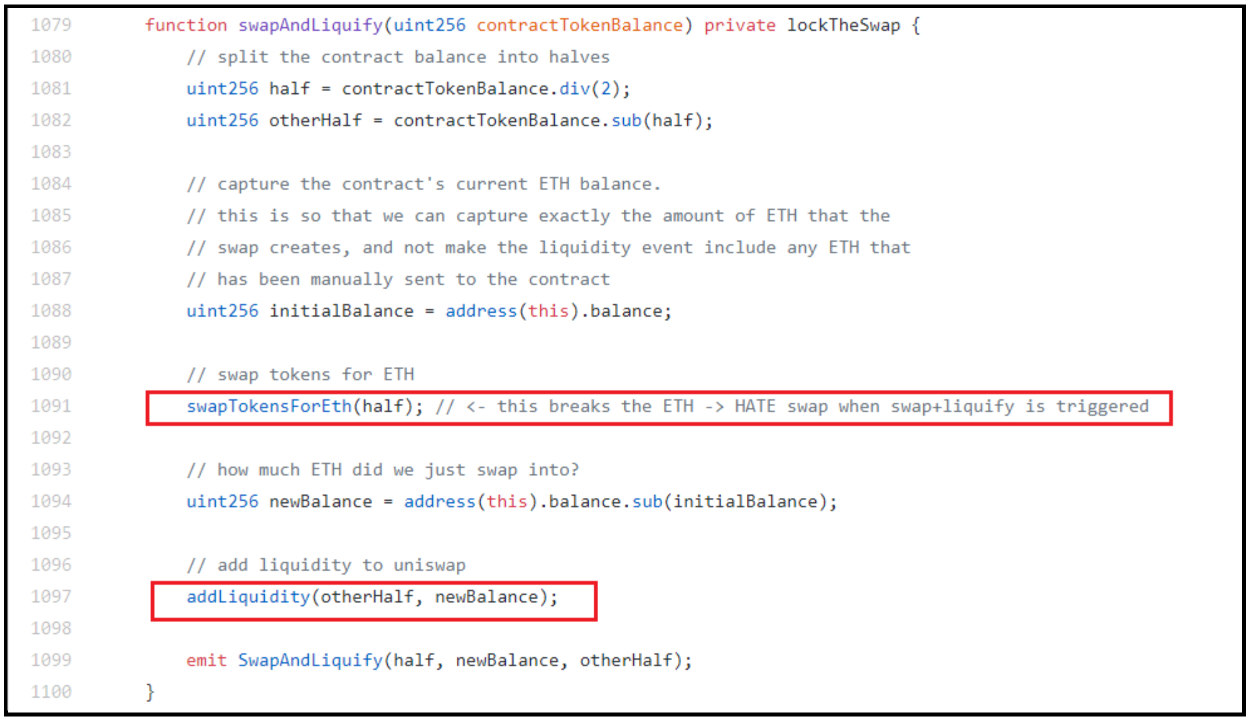
- Recommendation: Apply the check-effects-interactions pattern.

7. Reentrancy vulnerabilities

- Severity: Low
- Result: Found
- Confidence: Medium
- Bug Common Description: Detection of the reentrancy bug. Only report reentrancy that acts as a double call (see reentrancy-eth, reentrancy-no-eth).
- POC:
  External calls:
  - swapTokensForEth(half) (#1091)
  - uniswapV2Router.swapExactTokensForETHSupportingFeeOnTransferTokens(tokenAmount,0,path,address(this),block.timestamp) (#1111-1117)
  - addLiquidity(otherHalf,newBalance) (#1097)
    - uniswapV2Router.addLiquidityETH{value: ethAmount}(address(this),tokenAmount,0,0,owner(),block.timestamp) (#1125-1132)

  External calls sending eth:
  - addLiquidity(otherHalf,newBalance) (#1097)
  - uniswapV2Router.addLiquidityETH{value: ethAmount}(address(this),tokenAmount,0,0,owner(),block.timestamp) (#1125-1132)

  State variables written after the call(s):
  - addLiquidity(otherHalf,newBalance) (#1097)
  - _allowances[owner][spender] = amount (#1029)



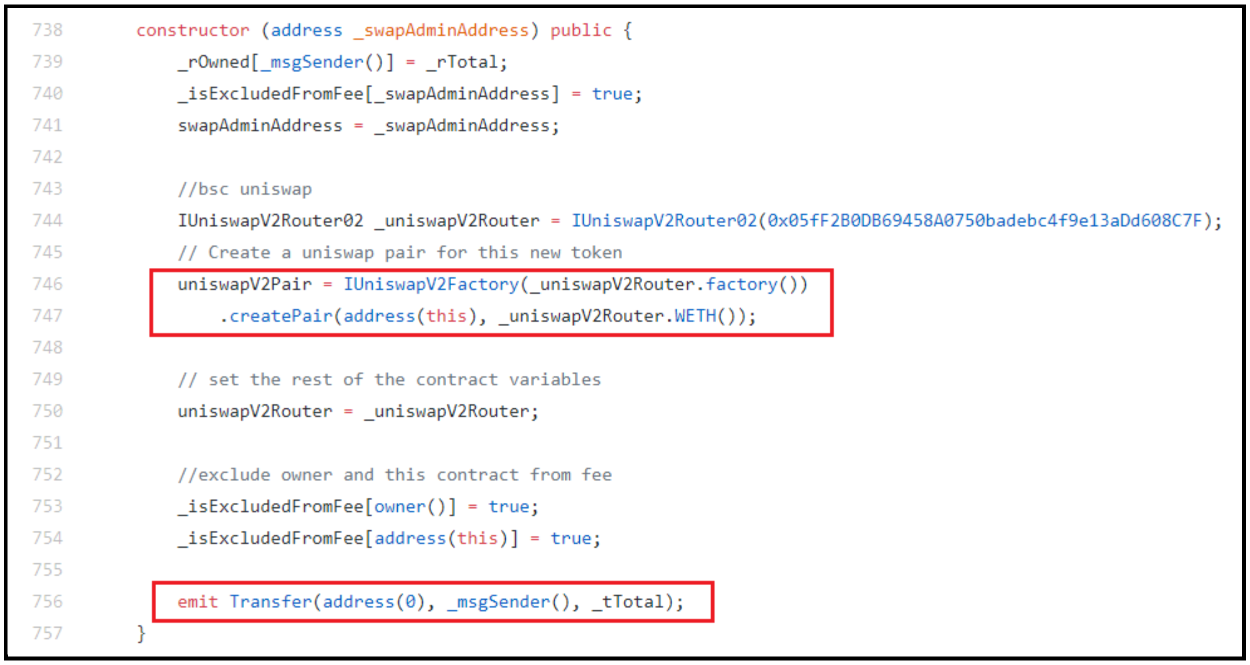
```
1079     function swapAndLiquify(uint256 contractTokenBalance) private lockTheSwap {
1080         // split the contract balance into halves
1081         uint256 half = contractTokenBalance.div(2);
1082         uint256 otherHalf = contractTokenBalance.sub(half);
1083 
1084         // capture the contract's current ETH balance.
1085         // this is so that we can capture exactly the amount of ETH that the
1086         // swap creates, and not make the liquidity event include any ETH that
1087         // has been manually sent to the contract
1088         uint256 initialBalance = address(this).balance;
1089 
1090         // swap tokens for ETH
1091         swapTokensForEth(half); // <- this breaks the ETH -> HATE swap when swap+liquify is triggered
1092 
1093         // how much ETH did we just swap into?
1094         uint256 newBalance = address(this).balance.sub(initialBalance);
1095 
1096         // add liquidity to uniswap
1097         addLiquidity(otherHalf, newBalance);
1098 
1099         emit SwapAndLiquify(half, newBalance, otherHalf);
1100     }
```

- Recommendation: Apply the check-effects-interactions pattern.

8. Reentrancy vulnerabilities

- Severity: Low
- Result: Found
- Confidence: Medium
- Bug Common Description: Detection of the reentrancy bug. Only report reentrancies leading to out-of-order events.
- POC:
  External calls:
  - uniswapV2Pair = IUniswapV2Factory(_uniswapV2Router.factory()).createPair(address(this),_uniswapV2Router.WETH()) (#746-747)

  Event emitted after the call(s):
  - Transfer(address(0),_msgSender(),_tTotal) (#756)



```
738     constructor (address _swapAdminAddress) public {
739         _rOwned[_msgSender()] = _rTotal;
740         _isExcludedFromFee[_swapAdminAddress] = true;
741         swapAdminAddress = _swapAdminAddress;
742
743         //bsc uniswap
744         IUniswapV2Router02 _uniswapV2Router = IUniswapV2Router02(0x05fF2B0DB69458A0750badebc4f9e13aDd608C7F);
745         // Create a uniswap pair for this new token
746         uniswapV2Pair = IUniswapV2Factory(_uniswapV2Router.factory())
747             .createPair(address(this), _uniswapV2Router.WETH());
748
749         // set the rest of the contract variables
750         uniswapV2Router = _uniswapV2Router;
751
752         //exclude owner and this contract from fee
753         _isExcludedFromFee[owner()] = true;
754         _isExcludedFromFee[address(this)] = true;
755
756         emit Transfer(address(0), _msgSender(), _tTotal);
757     }
```

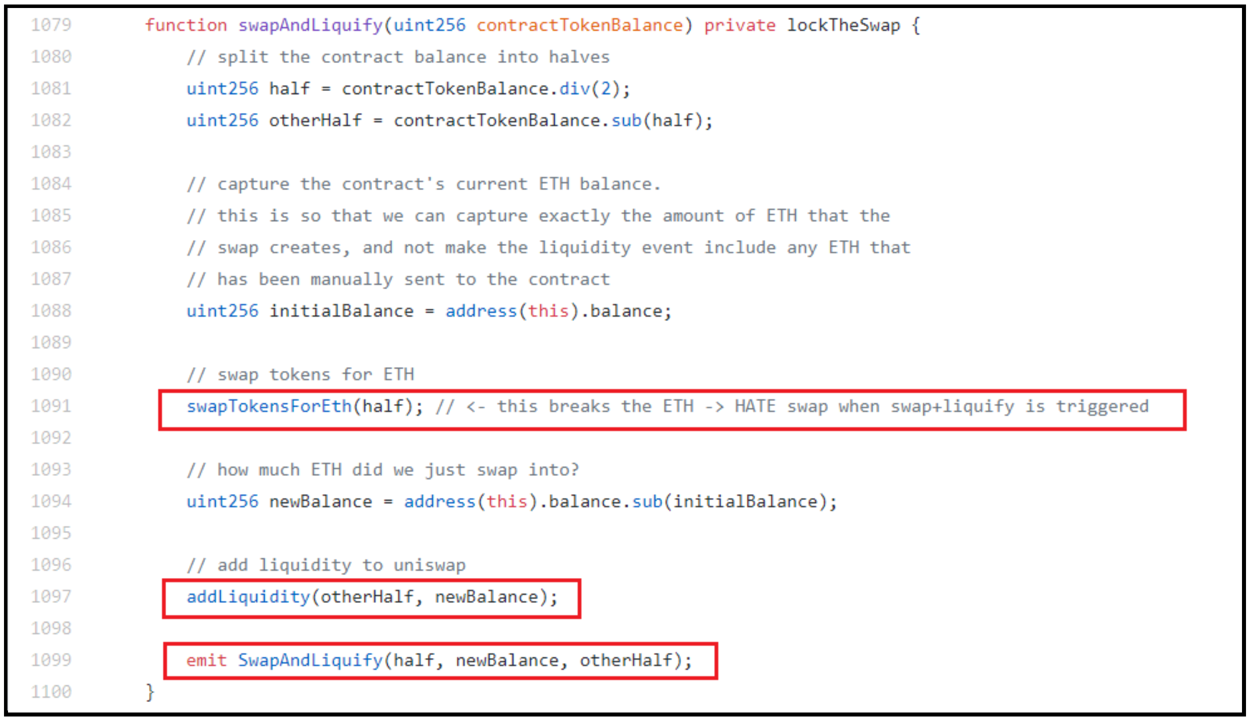
- Recommendation: Apply the check-effects-interactions pattern.

9. Reentrancy vulnerabilities

- Severity: Low
- Result: Found
- Confidence: Medium
- Bug Common Description: Detection of the reentrancy bug. Only report reentrancies leading to out-of-order events.
- POC:
  External calls:
  - swapTokensForEth(half) (#1091)
  - uniswapV2Router.swapExactTokensForETHSupportingFeeOnTransferTokens(tokenAmount,0,path,address(this),block.timestamp) (#1111-1117)
  - addLiquidity(otherHalf,newBalance) (#1097)
  - uniswapV2Router.addLiquidityETH{value: ethAmount}(address(this),tokenAmount,0,0,owner(),block.timestamp) (#1125-1132)

  External calls sending eth:
  - addLiquidity(otherHalf,newBalance) (#1097)
  - uniswapV2Router.addLiquidityETH{value: ethAmount}(address(this),tokenAmount,0,0,owner(),block.timestamp) (#1125-1132)

  Event emitted after the call(s):
  - Approval(owner,spender,amount) (#1030)
  - addLiquidity(otherHalf,newBalance) (#1097)
  - SwapAndLiquify(half,newBalance,otherHalf) (#1099)

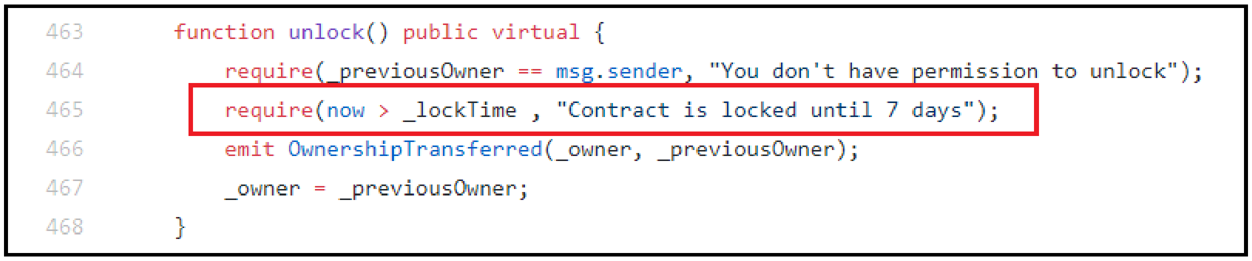


```
1079    function swapAndLiquify(uint256 contractTokenBalance) private lockTheSwap {
1080        // split the contract balance into halves
1081        uint256 half = contractTokenBalance.div(2);
1082        uint256 otherHalf = contractTokenBalance.sub(half);
1083
1084        // capture the contract's current ETH balance.
1085        // this is so that we can capture exactly the amount of ETH that the
1086        // swap creates, and not make the liquidity event include any ETH that
1087        // has been manually sent to the contract
1088        uint256 initialBalance = address(this).balance;
1089
1090        // swap tokens for ETH
1091        swapTokensForEth(half); // <- this breaks the ETH -> HATE swap when swap+liquify is triggered
1092
1093        // how much ETH did we just swap into?
1094        uint256 newBalance = address(this).balance.sub(initialBalance);
1095
1096        // add liquidity to uniswap
1097        addLiquidity(otherHalf, newBalance);
1098
1099        emit SwapAndLiquify(half, newBalance, otherHalf);
1100    }
```

- Recommendation: Apply the check-effects-interactions pattern.

10. Block timestamp

- Severity: Low
- Result: Found
- Confidence: Medium
- Bug Common Description: Dangerous usage of block.timestamp. block.timestamp can be manipulated by miners.
- POC:



```
function unlock() public virtual {
    require(_previousOwner == msg.sender, "You don't have permission to unlock");
    require(now > _lockTime , "Contract is locked until 7 days");
    emit OwnershipTransferred(_owner, _previousOwner);
    _owner = _previousOwner;
}
```

- Recommendation: Avoid relying on block.timestamp.

11. **Constructor Mismatch**

    - Bug Common Description: Whether the contract name and its constructor are not identical to each other.
    - Result: Not found
    - Severity: Critical

12. **Ownership Takeover**

    - Bug Common Description: Whether the set owner function is not protected.
    - Result: Not found
    - Severity: Critical

13. **Redundant Fallback Function**

    - Bug Common Description: Whether the contract has a redundant fallback function.
    - Result: Not found
    - Severity: Critical

14. **Overflows & Underflows**

    - Bug Common Description: Whether the contract has general overflow or underflow vulnerabilities
    - Result: Not found
    - Severity: Critical

15. **Reentrancy**

    - Bug Common Description: Reentrancy is an issue when code can call back into your contract and change state, such as withdrawing ETHs.
    - Result: Not found
    - Severity: Critical

16. **Money-Giving Bug**

    - Bug Common Description: Whether the contract returns funds to an arbitrary address.
    - Result: Not found
    - Severity: High

17. **Blackhole**

    - Bug Common Description: Whether the contract locks ETH indefinitely: merely in without out.
    - Result: Not found
    - Severity: High

18. **Unauthorized Self-Destruct**
    - Bug Common Description: Whether the contract can be killed by any arbitrary address.
    - Result: Not found
    - Severity: Medium

19. **Revert DoS**
    - Bug Common Description: Whether the contract is vulnerable to DoS attack because of unexpected revert.
    - Result: Not found
    - Severity: Medium

20. **Unchecked External Call**
    - Bug Common Description: Whether the contract has any external call without checking the return value.
    - Result: Not found
    - Severity: Medium

21. **Gasless Send**
    - Bug Common Description: Whether the contract is vulnerable to gasless send.
    - Result: Not found
    - Severity: Medium

22. **Send Instead of Transfer**
    - Bug Common Description: Whether the contract uses send instead of transfer.
    - Result: Not found
    - Severity: Medium

23. **Costly Loop**
    - Bug Common Description: Whether the contract has any costly loop which may lead to Out-Of-Gas exception.
    - Result: Not found
    - Severity: Medium

24. **(Unsafe) Use of Untrusted Libraries**
    - Bug Common Description: Whether the contract use any suspicious libraries.
    - Result: Not found
    - Severity: Medium

25. **(Unsafe) Use of Predictable Variables**

    - Bug Common Description: Whether the contract contains any randomness variable, but its value can be predicated.
    - Result: Not found
    - Severity: Medium

26. **Transaction Ordering Dependence**

    - Bug Common Description: Whether the final state of the contract depends on the order of the transactions.
    - Result: Not found
    - Severity: Medium

27. **Deprecated Uses**

    - Bug Common Description: Whether the contract use the deprecated tx.origin to perform the authorization.
    - Result: Not found
    - Severity: Medium

Semantic Consistency Checks

- Bug Common Description: Whether the semantic of the white paper is different from the implementation of the contract.
- Result: Not found
- Severity: Critical

Conclusion

In this audit, we thoroughly analyzed the Only Up’s ‘SafeMoon’ Smart Contract. The current code base is well organized but there are promptly some medium and low-level issues found in this phase of Smart Contract Audit.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

About eNebula Solutions

We believe that people have a fundamental need to security and that the use of secure solutions enables every person to more freely use the Internet and every other connected technology. We aim to provide security consulting service to help others make their solutions more resistant to unauthorized access to data & inadvertent manipulation of the system. We support teams from the design phase through the production to launch and surely after.

The eNebula Solutions team has skills for reviewing code in C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, and JavaScript for common security vulnerabilities & specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture, including incryptocurrency, blockchains, payments, and smart contracts. Additionally, the team can utilize varioustools to scan code & networks and build custom tools as necessary.

Although we are a small team, we surely believe that we can have a momentous impact on the world by being translucent and open about the work we do.

For more information about our security consulting, please
mail us at – contact@enebula.in.