SMART CONTRACT AUDIT REPORT

for

AngryToken

Prepared By: Yiqun Chen

PeckShield

June 28, 2021

Document Properties

Client	AngryToken
Title	Smart Contract Audit Report
Target	AngryToken
Version	1.0
Author	Xuxian Jiang
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author	Description
1.0	June 28, 2021	Xuxian Jiang	Final Release
1.0-rc	June 14, 2021	Xuxian Jiang	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the **AngryToken** smart contract, we outline in the report our systematic method to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistency between smart contract code and the documentation, and provide additional suggestions or recommendations for improvement. Our results show that the given version of the smart contract exhibits no ERC20 compliance issues, but carries with a number of important security concerns. This document outlines our audit results.

1.1 About AngryToken

`AngryToken` is a standard ERC20 token contract. This audit covers the ERC20-compliance of the given `AngryToken` token as well as the accompanying `AngryContract` that is designed to support token pre-purchase and reward-claiming.

The basic information of AngryToken is as follows:

Table 1.1: Basic Information of AngryToken

Item	Description
Name	AngryToken
Type	Ethereum ERC20 Token Contract
Platform	Solidity
Audit Method	Whitebox
Audit Completion Date	June 28, 2021

In the following, we show the compressed file with the source contract for audit and the MD5/SHA checksum value of the compressed file:

- Name: AngryToken.zip
- MD5: 1b0080f7420cead9b788d671c725674d

- SHA256: d006bde682d91b69d2c33f58501aac045b1fe0603dc50d9acaf49400d0dbc761

And this is the MD5 checksum of the compressed file after all fixes for the issues found in the audit have been checked in: 7fe8baad7ffd7eb056254bec79776042.

1.2 About PeckShield

PeckShield Inc. [10] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystem by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [9]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk;

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

We perform the audit according to the following procedures:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- ERC20 Compliance Checks: We then manually check whether the implementation logic of the audited smart contract(s) follows the standard ERC20 specification and other best practices.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.3: 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
	Approve / TransferFrom Race Condition
ERC20 Compliance Checks	Compliance Checks (Section 3)
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

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the AngryToken. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place ERC20-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	2	■■
Medium	1	■
Low	2	■■
Informational	0	
Total	5	

Moreover, we explicitly evaluate whether the given contracts follow the standard ERC20 specification and other known best practices, and validate its compatibility with other similar ERC20 tokens and current DeFi protocols. The detailed ERC20 compliance checks are reported in Section 3. After that, we examine a few identified issues of varying severities that need to be brought up and paid more attention to. (The findings are categorized in the above table.) Additional information can be found in the next subsection, and the detailed discussions are in Section 4.

2.2 Key Findings

Overall, no ERC20 compliance issue was found, detailed checklist can be found in Section 3. However, the smart contract implementation can be improved because of the existence of 2 high-severity vulnerabilities, 1 medium-severity vulnerability, and 2 low-severity vulnerabilities

Table 2.1: Key AngryToken Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Suggested ExecutorAdd Event in Constructor()	Coding Practices	Fixed
PVE-002	High	Possible Sandwich/MEV For Reduced Purchase Price	Time And State	Fixed
PVE-003	High	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time And State	Fixed
PVE-004	Low	Improved Sanity Checks For System Parameters	Coding Practices	Fixed
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for our detailed compliance checks and Section 4 for elaboration of reported issues.

3 ERC20 Compliance Checks

The ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20-compliant. Naturally, as the first step of our audit, we examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Table 3.1: Basic `View-Only` Functions Defined in The ERC20 Specification

Item	Description	Status
name()	Is declared as a public view function	✓
	Returns a string, for example "Tether USD"	✓
symbol()	Is declared as a public view function	✓
	Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length	✓
decimals()	Is declared as a public view function	✓
	Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required	✓
totalSupply()	Is declared as a public view function	✓
	Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment	✓
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
allowance()	Is declared as a public view function	✓
	Returns the amount which the spender is still allowed to withdraw from the owner	✓

Our analysis shows that there is no ERC20 inconsistency or incompatibility issue found in the audited AngryToken. In the surrounding two tables, we outline the respective list of basic `view-only` functions (Table 3.1) and key `state-changing` functions (Table 3.2) according to the widely-

Table 3.2: Key `State-Changing` Functions Defined in The ERC20 Specification

Item	Description	Status
transfer()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the caller does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring to zero address	✓
transferFrom()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the spender does not have enough token allowances to spend	✓
	Updates the spender's token allowances when tokens are transferred successfully	✓
	Reverts if the from address does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring from zero address	✓
	Reverts while transferring to zero address	✓
approve()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token approval status	✓
	Emits Approval() event when tokens are approved successfully	✓
	Reverts while approving to zero address	✓
Transfer() event	Is emitted when tokens are transferred, including zero value transfers	✓
	Is emitted with the from address set to $address(0x0)$ when new tokens are generated	✓
Approval() event	Is emitted on any successful call to approve()	✓

adopted ERC20 specification. In addition, we perform a further examination on certain features that are permitted by the ERC20 specification or even further extended in follow-up refinements and enhancements (e.g., ERC777/ERC2222), but not required for implementation. These features are generally helpful, but may also impact or bring certain incompatibility with current DeFi protocols. Therefore, we consider it is important to highlight them as well. This list is shown in Table 3.3.

Table 3.3: Additional `Opt-in` Features Examined in Our Audit

Feature	Description	Opt-in
Deflationary	Part of the tokens are burned or transferred as fee while on transfer()/transferFrom() calls	—
Rebasing	The balanceOf() function returns a re-based balance instead of the actual stored amount of tokens owned by the specific address	—
Pausable	The token contract allows the owner or privileged users to pause the token transfers and other operations	—
Blacklistable	The token contract allows the owner or privileged users to blacklist a specific address such that token transfers and other operations related to that address are prohibited	—
Mintable	The token contract allows the owner or privileged users to mint tokens to a specific address	—
Burnable	The token contract allows the owner or privileged users to burn tokens of a specific address	✓

4 | Detailed Results

4.1 Suggested ExecutorAdd Event in Constructor()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: AngryContract
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

In Ethereum, the event is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an event is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the AngryContract contract as an example. While examining the events that reflect the AngryContract token dynamics, we notice there is a lack of emitting important events that reflect important state changes. Specifically, when the very first member of executorList is being added, there is no respective event being emitted to reflect the dynamics.

```
    constructor (address _angryTokenAddr, address _usdtTokenAddr, address
        _uniswapRouterAddr, address _vitalikButerinAddr){
        owner = msg.sender;
        executorList[msg.sender] = true;
        startTime = block.timestamp;
        angryTokenAddr = _angryTokenAddr;
        usdtTokenAddr = _usdtTokenAddr;
        uniswapRouterAddr = _uniswapRouterAddr;
        vitalikButerinAddr = _vitalikButerinAddr;
        angryToken = IERC20(angryTokenAddr);
        usdtToken = IERC20(usdtTokenAddr);
        uniswapRouterV2 = IUniswapV2Router02(uniswapRouterAddr);
        angryTokenDecimals = IERC20Metadata(angryTokenAddr).decimals();
```

```
    }
```

Listing 4.1: `AngryContract::constructor()`

Moreover, the `VBRewardBurn` event is better relocated into the `then`-branch. Currently, it is emitted outside the `if`-clause.

Recommendation Properly emit the `ExecutorAdd` event when the very first `executorList` is added. This is very helpful for external analytics and reporting tools.

Status The issue has been confirmed and accordingly fixed by adding the suggested events.

4.2 Possible Sandwich/MEV For Reduced Purchase Price

- ID: PVE-002
- Severity: High
- Likelihood: High
- Impact: High
- Target: `AngryContract`
- Category: Time and State [8]
- CWE subcategory: CWE-682 [4]

Description

As mentioned earlier, the `AngryToken` contract supports pre-sale that allows early adopters to purchase the token. While examining the pre-sale support, we notice the purchase price can be arbitrarily manipulated.

To elaborate, we show below the `prePurchase()` function. As the name indicates, it is designed to implement the pre-sale functionality. It comes to our attention that the purchase price is computed on-chain via an internal helper `getANBPrice()`.

```
    function prePurchase(uint256 _expectedPrice, uint256 _startTime, uint256
        _expiredTime) public payable {
        require( _expiredTime > _startTime, "Incorrect time period!" );
        uint256 accountQuota = getAccountPurchaseQuota(msg.sender);
        require( accountQuota > 0, "Exceed account quota!" );
        uint256 currAmount = 0;
        PrePurchaseInfo[] storage purchases = prePurchaseList[ msg.sender ];
        PrePurchaseInfo memory pcInfo;
        uint256 ethPrice = 0;
        uint256 usdtPrice = 0;
        (ethPrice, usdtPrice) = getANBPrice();
        if(msg.value > 0){
            require(ethPrice > 0, "Invalid ethPrice!");
            uint256 highestEthPrice = ethPrice * maxPriceMultiple * (100 +
                expectedPriceFloatVal) / 100;
            require( _expectedPrice <= highestEthPrice, "expectedPrice too high!" );
            currAmount = msg.value * 10 ** angryTokenDecimals / ethPrice;
```

```
            pcInfo.price = ethPrice;
            pcInfo.paymentAmount = msg.value;
            pcInfo.paymentType = 1;
        }else{
            require(usdtPrice > 0, "Invalid usdtPrice!");
            uint256 highestUSDTPrice = usdtPrice * maxPriceMultiple * (100 + expectedPriceFloatVal) / 100;
            require( _expectedPrice <= highestUSDTPrice, "expectedPrice too high!" );
            uint256 allowance = usdtToken.allowance(msg.sender, address(this));
            require( allowance > 0, "Not any payments!" );
            currAmount = allowance * 10 ** angryTokenDecimals / usdtPrice;
            pcInfo.price = usdtPrice;
            usdtToken.safeTransferFrom(
                msg.sender,
                address(this),
                allowance
            );
            pcInfo.paymentAmount = allowance;
            pcInfo.paymentType = 2;
        }
        uint256 totalQuota = queryCurrPrePurchaseQuota();
        require( (currAmount + totalPrePurcaseAmount) <= totalQuota, "Exceed daily quota!" );
        require( currAmount <= accountQuota, "Exceed account quota!" );
        if(purchases.length == 0){
            prePurchaseAccounts.push(msg.sender);
        }
        pcInfo.amount = currAmount;
        pcInfo.expectedPrice = _expectedPrice;
        pcInfo.startTime = _startTime;
        pcInfo.expiredTime = _expiredTime;
        pcInfo.status = 0;
        purchases.push(pcInfo);
        totalPrePurcaseAmount = totalPrePurcaseAmount + currAmount;
        emit PrePurchase(msg.sender, purchases.length-1, currAmount, pcInfo.paymentAmount, pcInfo.price, _expectedPrice, _startTime, _expiredTime, pcInfo.paymentType, pcInfo.status);
    }
```

Listing 4.2: `AngryContract::prePurchase()`

```
    function getANBPrice() public view returns(uint256 _ethPrice, uint256 _usdtPrice){
        address[] memory path = new address[](3);
        path[0] = angryTokenAddr;
        path[1] = uniswapRouterV2.WETH();
        path[2] = usdtTokenAddr;
        uint256[] memory amounts = uniswapRouterV2.getAmountsOut(10 ** angryTokenDecimals, path);
        _ethPrice = amounts[1];
        _usdtPrice = amounts[2];
    }
```

Listing 4.3: `AngryContract::getANBPrice()`

Specifically, the purchase price is directly returned by querying the trading price of `UniswapV2` on the trading path `ANB -> WETH -> USDT` without imposing any restriction. As a result, the current pricing approach is vulnerable to possible sandwich attacks, resulting in a manipulated purchase price.

A similar issue is also present in `processPrePurchaseOrder()`, which does not have any slippage control in place.

Recommendation Develop an effective mitigation to the above sandwich attack to better protect the interests of purchasing users.

Status The issue has been fixed by specifying necessary slippage control in related functions.

4.3 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `AngryContract`
- Category: Time and State [7]
- CWE subcategory: CWE-663 [3]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [12] exploit, and the recent `Uniswap/Lendf.Me` hack [11].

We notice there are several occasions where the `checks-effects-interactions` principle is violated. Using the `AngryContract` as an example, the `cancelPrePurchaseOrder()` function (see the code snippet below) is provided to call an external (untrusted) address to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 375) starts before effecting the update on the internal state (line 380), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the same entry function.

```
360     function cancelPrePurchaseOrder(uint256 _orderIdx) public {
361         PrePurchaseInfo[] storage purchases = prePurchaseList[ msg.sender ];
```

```
        require( purchases.length > _orderIdx, "Order index out of range!" );
        PrePurchaseInfo storage pcInfo = purchases[_orderIdx];
        require( pcInfo.status == 0 || pcInfo.status == 4, "Unexpected order status!" );
        uint256 fee = 0;
        uint256 refundAmount = pcInfo.paymentAmount;
        if(cancelOrderFeeRate > 0){
            fee = pcInfo.paymentAmount * cancelOrderFeeRate / 100000;
        }
        if(fee > 0){
            refundAmount = refundAmount - fee;
        }
        if(pcInfo.paymentType == 1){
            feeETH = feeETH + fee;
            payable(msg.sender).transfer(refundAmount);
        }else{
            feeUSDT = feeUSDT + fee;
            usdtToken.safeTransfer(msg.sender, refundAmount);
        }
        pcInfo.status = 2;
        emit OrderCancel(msg.sender, _orderIdx, pcInfo.status);
    }
```

Listing 4.4: `AngryContract::cancelPrePurchaseOrder()`

Note a similar issue is also present in other routines, including `withdrawRevenueAndFee()` and `processPrePurchaseOrder()`. The adherence of `checks-effects-interactions` best practice or the use of `nonReentrant` modifier is strongly recommended.

Recommendation Apply necessary reentrancy prevention by utilizing the `nonReentrant` modifier to block possible `re-entrancy`.

Status The issue has been addressed by placing the `nonReentrant` modifier with the affected functions.

4.4 Improved Sanity Checks For System/Function Parameters

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `AngryContract`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `AngryToken` is no exception. Specifically, if we examine the `AngryContract`

contract, it has defined a number of protocol-wide risk parameters, such as `maxMiningTaskReward` and `prePurchaseSupplyPerDay`. In the following, we show the corresponding routines that allow for their changes.

```
    function setMaxMiningTaskReward(uint256 _newValue) public onlyExecutor {
        emit MaxMiningTaskRewardChange(maxMiningTaskReward, _newValue);
        maxMiningTaskReward = _newValue;
    }

    function setPrePurchaseSupplyPerDay(uint256 _newValue) public onlyExecutor {
        emit PrePurchaseSupplyPerDayChange(prePurchaseSupplyPerDay, _newValue);
        prePurchaseSupplyPerDay = _newValue;
    }

    function setVbWithdrawPerDay(uint256 _newValue) public onlyExecutor {
        emit VbWithdrawPerDayChange(vbWithdrawPerDay, _newValue);
        vbWithdrawPerDay = _newValue;
    }
```

Listing 4.5: A number of representative `setters` in `AngryContract`

```
    function setPrePurchaseaArgs(uint256 _minTokenAmount, uint256 _maxMultiple, uint256
        _limitPerAcc) public onlyExecutor {
        emit PrePurchaseaArgsChange(minTokenAmountToPrePurchase,maxPrePurchaseMultiple,
            prePurchaseLimitPerAcc,_minTokenAmount,_maxMultiple,_limitPerAcc);
        minTokenAmountToPrePurchase = _minTokenAmount;
        maxPrePurchaseMultiple = _maxMultiple;
        prePurchaseLimitPerAcc = _limitPerAcc;
    }
```

Listing 4.6: `AngryContract::setPrePurchaseaArgs()`

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the parameter-setting logic (see the above `setPrePurchaseaArgs()` on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of `minTokenAmountToPrePurchase` may charge unreasonably high entry barrier in the `prePurchase()` operation, hence incurring cost to users or hurting the adoption of the protocol.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. If necessary, also consider emitting relevant events for their changes.

Status This issue has been fixed by following the above suggestion.

4.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `AngryContract`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the associated `AngryContract`, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configuring various system parameters). In the following, we show representative privileged operations in the protocol's core `AngryContract` contract.

```
    function withdrawANB(address _receiver, uint256 _amount) public onlyOwner {
        angryToken.safeTransfer(_receiver, _amount);
        emit ANBWithdraw(_receiver, _amount);
    }
```

Listing 4.7: `AngryContract::withdrawANB()`

```
    function setPrePurchaseaArgs(uint256 _minTokenAmount, uint256 _maxMultiple, uint256
        _limitPerAcc) public onlyExecutor {
        emit PrePurchaseaArgsChange(minTokenAmountToPrePurchase,maxPrePurchaseMultiple,
            prePurchaseLimitPerAcc,_minTokenAmount,_maxMultiple,_limitPerAcc);
        minTokenAmountToPrePurchase = _minTokenAmount;
        maxPrePurchaseMultiple = _maxMultiple;
        prePurchaseLimitPerAcc = _limitPerAcc;
    }
```

Listing 4.8: `AngryContract::setPrePurchaseaArgs()`

We emphasize that the privilege assignment may be necessary and consistent with the token design. However, it is worrisome if the `owner` is not governed by a `DAO`-like structure. Note that a compromised `owner` account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the `AngryToken` design.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed.

5 Conclusion

In this security audit, we have examined the AngryToken design and implementation. During our audit, we first checked all respects related to the compatibility of the ERC20 specification and other known ERC20 pitfalls/vulnerabilities. We then proceeded to examine other areas such as coding practices and business logics. Overall, we have identified several high-severity issues that require prompt attention and urgent fixes from the team. In the meantime, as disclaimed in Section 1.4, we appreciate any constructive feedbacks or suggestions about our findings, procedures, audit scope, etc.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[4] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[5] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[6] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[7] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

[8] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[9] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[10] PeckShield. PeckShield Inc. https://www.peckshield.com.

[11] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[12] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.