SMART CONTRACT AUDIT REPORT

for

AngryMining

Prepared By: Yiqun Chen

PeckShield

July 12, 2021

Document Properties

Client	AngryToken
Title	Smart Contract Audit Report
Target	AngryMining
Version	1.0-rc
Author	Jing Wang
Auditors	Xuxian Jiang, Jing Wang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc	July 12, 2021	Jing Wang	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 smart contract source code of the `AngryMining` protocol, we outline in the report 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 can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About AngryMining

The `AngryMining` is a decentralized liquidity mining platform which provides a incentive mechanism to reward the staking of supported assets with certain reward tokens. The `LP` provider could deposit funds into liquidity pool and earn rewards in return.

The basic information of the `AngryMining` protocol is as follows:

Table 1.1: Basic Information of The `AngryMining` Protocol

Item	Description
Issuer	AngryToken
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 12, 2021

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

- Name: angryMining(2021.7.5).zip
- MD5: 8aec81822b9e3bd9198987a2913eb8b2

- SHA256: 0bb4a3571d68a489077284f9340bd3836c566136311e01065afe22480e225178

And this is the MD5 checksum of the compressed file after all fixes for the issues found in the audit have been checked in: 7bbf118a4667aff33f5e6a3ead869438.

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems 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).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- 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.

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
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	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

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 or analysis 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.

In particular, we perform the audit according to the following procedure:

- 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.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

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 1.4: 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 security software.)
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/status codes 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. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the AngryMining implementation. 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 DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	🟦
Low	2	🟦 🟦
Informational	1	🟦
Total	4	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key AngryMining Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Duplicate Pool Detection and Prevention	Business Logic	Fixed
PVE-002	Low	Improved Validation Of Function Arguments	Coding Practices	Fixed
PVE-003	Informational	Recommended Explicit Pool Validity Checks	Security Features	Fixed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 details.

3 Detailed Results

3.1 Duplicate Pool Detection and Prevention

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: AngryMining
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The AngryMining protocol provides incentive mechanisms that reward the staking of supported assets with certain reward tokens. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. Each pool has its allocPoint*100%/totalAllocPoint share of scheduled rewards and the rewards for stakers are proportional to their share of LP tokens in the pool.

In current implementation, there are a number of concurrent pools that share the rewarded tokens and more can be scheduled for addition. To accommodate these new pools, the design has the necessary mechanism in place that allows for dynamic additions of new staking pools that can participate in being incentivized as well.

The addition of a new pool is implemented in addPool(), whose code logic is shown below. It turns out it did not perform necessary sanity checks in preventing a new pool but with a duplicate token from being added. Though it is a privileged interface (protected with the modifier onlyExecutor), it is still desirable to enforce it at the smart contract code level, eliminating the concern of wrong pool introduction from human omissions.

```
    function addPool(
        uint256 _allocPoint,
        address _lpToken
    ) public onlyExecutor {
        massUpdatePools();
        totalAllocPoint = totalAllocPoint + _allocPoint;
        poolInfo.push(
            PoolInfo({
                lpToken: IERC20(_lpToken),
                allocPoint: _allocPoint,
                lastRewardBlock: block.number,
                accAngryPerShare: 0
            })
        );
        emit PoolAdd(_allocPoint,_lpToken);
    }
```

Listing 3.1: `AngryMining::addPool()`

Recommendation Detect whether the given pool for addition is a duplicate of an existing pool. The pool addition is only successful when there is no duplicate.

```
    function checkPoolDuplicate(IERC20 _lpToken) public {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            require(poolInfo[_pid].lpToken != _lpToken, "add: existing pool?");
        }
    }

    function addPool(
        uint256 _allocPoint,
        address _lpToken
    ) public onlyExecutor {
        massUpdatePools();
        checkPoolDuplicate(_lpToken);
        totalAllocPoint = totalAllocPoint + _allocPoint;
        poolInfo.push(
            PoolInfo({
                lpToken: IERC20(_lpToken),
                allocPoint: _allocPoint,
                lastRewardBlock: block.number,
                accAngryPerShare: 0
            })
        );
        emit PoolAdd(_allocPoint,_lpToken);
    }
```

Listing 3.2: `AngryMining::addPool()`

We point out that if a new pool with a duplicate LP token can be added, it will likely cause a havoc in the distribution of rewards to the pools and the stakers.

Status This issue has been fixed by adding the function to check whether the given pool for addition is a duplicate of an existing pool.

3.2 Improved Validation Of Function Arguments

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: AngryMining
- Category: Coding Practices [5]
- CWE subcategory: CWE-628 [2]

Description

In the AngryMining contract, the addBonusPeriod() function is used to add a bonus period and give additional rewards from _beginBlock to _endBlock. To elaborate, we show below the related code snippet.

```
    function addBonusPeriod(uint256 _beginBlock, uint256 _endBlock) public onlyExecutor
        {
        uint256 length = bonusPeriods.length;
        for(uint256 i = 0;i < length; i++){
            require(_endBlock < bonusPeriods[i].beginBlock  _beginBlock > bonusPeriods[i
                ].endBlock, "BO");
        }
        massUpdatePools();
        BonusPeriod memory bp;
        bp.beginBlock = _beginBlock;
        bp.endBlock = _endBlock;
        bonusPeriods.push(bp);
        emit BonusPeriodAdd(_beginBlock, _endBlock);
    }
```

Listing 3.3: AngryMining::addBonusPeriod()

We notice that this function has an assumption that _beginBlock is less than _endBlock. However, there is no actual enforcement of this assumption in current implementation. If an invalid bonus period is added into the bonusPeriods array, it will cause unexpected errors in the getMultiplier() function.

Recommendation Make the requirement of require(_beginBlock < _endBlock) explicitly in AngryMining::addBonusPeriod().

Status This issue has been fixed by adding the suggested requirement into AngryMining::addBonusPeriod().

3.3 Recommended Explicit Pool Validity Checks

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: AngryMining
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

The AngryMining protocol provides the functionalities of the pool management, the staking/unstaking support, and the reward distribution to various pools and stakers. In the following, we show the key pool data structure. Note all added pools are maintained in an array poolInfo.

```
    // Info of each pool.
    struct PoolInfo {
        IERC20 lpToken; // Address of LP token contract.
        uint256 allocPoint; // How many allocation points assigned to this pool. ANGRYs
            to distribute per block.
        uint256 lastRewardBlock; // Last block number that ANGRYs distribution occurs.
        uint256 accAngryPerShare; // Accumulated ANGRYs per share, times 1e12. See below
            .
    }
    ...
    // Info of each pool.
    PoolInfo[] public poolInfo;
```

Listing 3.4: The PoolInfo Data Structure in AngryMining

When there is a need to add a new pool, set a new allocPoint for an existing pool, stake (by depositing the supported assets), unstake (by redeeming previously deposited assets), query pending rewards, there is a constant need to perform sanity checks on the pool validity. The current implementation simply relies on the implicit, compiler-generated bound-checks of arrays to ensure the pool index stays within the array range [0, poolInfo.length-1]. However, considering the importance of validating given pools and their numerous occasions, a better alternative is to make explicit the sanity checks by introducing a new modifier, say validatePool. This new modifier essentially ensures the given _pid indeed points to a valid, live pool, and additionally give semantically meaningful information when it is not!

```
    function depositLP(uint256 _pid, uint256 _amount) public nonReentrant {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pending =
                user.amount * pool.accAngryPerShare / (1e12) - user.rewardDebt;
            //safeAngryTransfer(msg.sender, pending);
            user.reward += pending;
        }
        pool.lpToken.safeTransferFrom(
            address(msg.sender),
            address(this),
            _amount
        );
        user.amount = user.amount + _amount;
        user.rewardDebt = user.amount * pool.accAngryPerShare / (1e12);
        emit LpDeposit(msg.sender, _pid, _amount);
    }
```

Listing 3.5: `AngryMining::depositLP()`

We highlight that there are a number of functions that can be benefited from the new pool-validating modifier, including `getLpMiningReward()`, `depositLP()`, `withdrawLP()`, `harvestLpMiningReward()`, `changePool()` and `updatePool()`.

Recommendation Apply necessary sanity checks to ensure the given `_pid` is legitimate. Accordingly, a new modifier `validatePool` can be developed and appended to each function in the above list.

```
     modifier validatePool(uint256 _pid) {
        require(_pid < poolInfo.length, "chef: pool exists?");
        _;
    }
```

Listing 3.6: The New `validatePool()` Modifier

Status This issue has been fixed by following the above suggestion to add the `validatePool()` modifier.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `AngryMining`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the `AngryMining` protocol, there is a special administrative account, i.e., `executor`. This `executor` account plays a critical role in governing and regulating the system-wide operations (e.g., pool addition, reward adjustment). Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

To elaborate, we show below the `changePool()` function in the `AngryMining` contract. This function allows the `executor` to change a key factor, `allocPoint`, which greatly affects on how many shares of the LP pools could receive. What's more, due to the lack of constraint of the changed `_allocPoint`, the privileged account could even set the `_allocPoint` to 0, which leads to no rewards received from the related LP pool.

```
    function changePool(
        uint256 _pid,
        uint256 _allocPoint
    ) public onlyExecutor {
        massUpdatePools();
        totalAllocPoint = totalAllocPoint - poolInfo[_pid].allocPoint + _allocPoint;
        poolInfo[_pid].allocPoint = _allocPoint;
        emit PoolChange(_pid, _allocPoint);
    }
```

Listing 3.7: `AngryMining::changePool()`

It is worrisome if the privileged `executorList` account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

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 The team has mitigated this issue by appending a multi-sig modifier to `changePool()` and `addPool()`. Also, a requirement of `require( _allocPoint > 0)` has been added into the function `changePool()` to make sure the `_allocPoint` is larger than 0.

4 Conclusion

In this audit, we have analyzed the `AngryMining` protocol design and implementation. The `AngryMining` protocol provides a decentralized liquidity platform for `LP` provider to deposit funds into liquidity pool and earn rewards in return. During the audit, we notice that the current code base is well organized and those identified issues are promptly confirmed and fixed.

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.

References

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

[2] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.html.

[3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

[6] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[7] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

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

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