SMART CONTRACT AUDIT REPORT

for

Binopoly

Prepared By: Yiqun Chen

PeckShield
November 7, 2021

Document Properties

Client	Binopoly
Title	Smart Contract Audit Report
Target	Binopoly
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Yiqun Chen, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 7, 2021	Xuxian Jiang	Final Release
1.0-rc	October 21, 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 4
- 1.1 About Binopoly 4
- 1.2 About PeckShield 5
- 1.3 Methodology 5
- 1.4 Disclaimer 7

2 Findings 9
- 2.1 Summary 9
- 2.2 Key Findings 10

3 Detailed Results 11
- 3.1 Improved Corner Case Handling in BinoToken 11
- 3.2 Possible DoS With _getRandomMultiplier() Pre-Validation 12
- 3.3 Improved levelUp() Fairness in BrickFactory 14
- 3.4 Suggested Adherence Of Checks-Effects-Interactions Pattern 15
- 3.5 Proper pendingRewards() Calculation in BinoDistributionLaw 17
- 3.6 Timely massUpdatePools During Pool Weight Changes 18
- 3.7 Duplicate Pool Detection and Prevention 20
- 3.8 Staking Incompatibility With Deflationary Tokens 22

4 Conclusion 25

References 26

1 Introduction

Given the opportunity to review the design document and related smart contract source code of `Binopoly`, 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 Binopoly

`Binopoly` is a play-to-earn game to introduce the crypto finance concept to everyone, using a similar approach from the most popular financial educational board game in history. Players will be able to own properties and lands to collect rent and fees, which will be NFTs that could be sold on the `BinoMarketplace`. All of this will be integrated with our real-world geography of continents, countries and cities. The basic information of `Binopoly` is as follows:

Table 1.1: Basic Information of `Binopoly`

Item	Description
Name	Binopoly
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	November 7, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/zem007/Bino_Houses_NFT_Project.git (c7aa698)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/zem007/Bino_Houses_NFT_Project.git (6ce47bb)

1.2 About PeckShield

PeckShield Inc. [7] 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 [6]:

- 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.

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

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

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) [5], 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 `Binopoly` 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	2	■■
Low	5	■■■■■
Informational	0	
Undetermined	1	■
Total	8	

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 2 medium-severity vulnerabilities, 5 low-severity vulnerabilities, and 1 undetermined issue.

Table 2.1: Key Binopoly Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Corner Case Handling in BinoToken	Business Logic	Confirmed
PVE-002	Medium	Possible DoS With _getRandomMultiplier() Pre-Validation	Business Logic	Confirmed
PVE-003	Low	Improved levelUp() Fairness in BrickFactory	Business Logic	Resolved
PVE-004	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	Resolved
PVE-005	Medium	Proper pendingRewards() Calculation in BinoDistributionLaw	Business Logic	Resolved
PVE-006	Low	Timely massUpdatePools During Pool Weight Changes	Business Logic	Resolved
PVE-007	Low	Duplicate Pool Detection and Prevention	Business Logic	Resolved
PVE-008	Undetermined	Staking Incompatibility With Deflationary Tokens	Business Logic	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 details.

3 Detailed Results

3.1 Improved Corner Case Handling in BinoToken

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BinoToken`
- Category: Business Logic [3]
- CWE subcategory: CWE-841 [2]

Description

The `BinoToken` follows the standardized ERC-20 specification with the extension to support snapshots. These snapshots allow for query of an account balance as well as `totalSupply` at any previous snapshot. While reviewing the snapshot-based query logic, we notice the current implementation of `balanceOfAt()` and `totalSupplyAt()` may be improved. In the following, we examine these two query functions.

To elaborate, we show below their implementation. While the current implementation is largely sound and correct, there is a corner case that can be improved. Specifically, snapshot ids are designed to increase monotonically, with the first value being 1. In other words, an id of 0 would be considered as invalid. However, if there is no snapshot being made, the current token does not support the query of a user balance and current `totalSupply`! With that, we suggest to extend these two functions to explicitly handle the case when the given `snapshotId` is 0.

```
    /**
     * @dev Retrieves the balance of `account` at the time `snapshotId` was created.
     */
    function balanceOfAt(address account, uint256 snapshotId) public view returns (
        uint256) {
        (bool snapshotted, uint256 value) = _valueAt(snapshotId,
            _accountBalanceSnapshots[account]);

        return snapshotted ? value : balanceOf(account);
    }

    /**
```

```
    * @dev Retrieves the total supply at the time `snapshotId` was created.
    */
   function totalSupplyAt(uint256 snapshotId) public view returns(uint256) {
       (bool snapshotted, uint256 value) = _valueAt(snapshotId, _totalSupplySnapshots);

       return snapshotted ? value : totalSupply();
   }
```

Listing 3.1: BinoToken::balanceOfAt()/totalSupplyAt()

Recommendation Improve the above two functions to handle the corner case where `snapshotId ==0`.

Status This issue has been confirmed.

3.2 Possible DoS With _getRandomMultiplier() Pre-Validation

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: `HousesNFT`, `BrickFactory`
- Category: Business Logic [3]
- CWE subcategory: CWE-841 [2]

Description

`Binopoly` has an essential `BrickFactory` contract to allow for tokenized materials and their assembly. During the examination of the logic to claim upgraded materials, we observe the randomness factor that is designed to compute the success ratio (via `_getRandomSuccessRate()`). However, this randomness computation may not be truly randomized and the caller may take advantage of it to choose to not claim the upgraded materials if the success ratio is not desired.

In the following, we use the related `claimUpgratedMaterial()` function. It has a rather straightforward logic and makes use of the helper routine `_getRandomSuccessRate()` to compute the `upgratedAmount` (line 326 and 334). However, the helper routine can be readily pre-computed to decide whether the success ratio is desired. If not, the caller may choose not to claim.

```
312     function claimUpgratedMaterial(uint256 lineId) public {
313         require(lineId != 0, "can not process line #0");
314         FactoryAttributes storage thisLevelAttributes = _currentAttributes[currentLevel
                ];
315         require(thisLevelAttributes.productionLineLimit >= lineId, "exceed production
                line limit");
316         require(isReadyToClaim(lineId), "not ready to claim now, try later");

318         address lineOwner = _linesUsage.get(lineId);
```

```
        uint256 finishTime = _completeTime[lineId];
        uint256 processAmount = _processAmounts[lineId];
        uint256 upgratedAmount;
        if (_msgSender() == lineOwner) {
            uint256 baseRate = thisLevelAttributes.baseSuccessRate;
            uint256 claimRate = _getRandomSuccessRate(baseRate);
            require(claimRate <= 10000, "claimRate exceed 100%");
            upgratedAmount = processAmount.mul(claimRate).div(10000);

            materialsAddress.mint(lineOwner, advanceMaterialId, upgratedAmount, "");
        } else {
            require(block.timestamp > finishTime.add(PROTECTION_PERIOD), "can not claim
                within protection time");
            uint256 baseRate = thisLevelAttributes.baseSuccessRate;
            uint256 claimRate = _getRandomSuccessRate(baseRate);
            require(claimRate <= 10000, "claimRate exceed 100%");
            upgratedAmount = processAmount.mul(claimRate).div(10000);
            uint256 shareAmount = upgratedAmount.div(10);              // 10%
            uint256 ownerAmount = upgratedAmount.sub(shareAmount);     // 90%

            materialsAddress.mint(lineOwner, advanceMaterialId, ownerAmount, "");
            materialsAddress.mint(_msgSender(), advanceMaterialId, shareAmount, "");
        }
        // remove and set to 0
        _linesUsage.remove(lineId);
        _completeTime[lineId] = 0;
        _processAmounts[lineId] = 0;

        emit ClaimUpgratedMaterial(lineId, lineOwner, upgratedAmount);
    }
```

Listing 3.2: BrickFactory::claimUpgratedMaterial()

```
    // generate a random integer between 0.9*base to 1.1*base (upperBound can not exceed
        10000)
    function _getRandomSuccessRate(uint256 base) private view returns (uint256) {
        // +/- 10%
        uint256 halfRange = base.div(10);
        uint256 lowerBound = base.sub(halfRange);
        uint256 upperBound = base.add(halfRange) >= 10000 ? 10000 : base.add(halfRange);
        // randomSeed % (upper - lower + 1) => randomInt = [0, upper - lower]
        uint256 randomInt = uint256(
            keccak256(
                abi.encodePacked(
                    uint256(blockhash(block.number.sub(1))),
                    uint256(block.coinbase),
                    block.difficulty,
                    block.timestamp,
                    base
                )
            )
        ).mod(upperBound.sub(lowerBound).add(1));
        return randomInt.add(lowerBound);
```

```
    }
```

Listing 3.3: `BrickFactory::_getRandomSuccessRate()`

In addition, the `HousesNFT` contract has a similar `_getRandomMultiplier()` routine that can be similarly computed to block possible `safeMint()` operations.

Recommendation Improve the randomness design in the above two functions and prevent them from being exploited.

Status This issue has been confirmed.

3.3 Improved levelUp() Fairness in BrickFactory

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `BrickFactory`
- Category: Business Logic [3]
- CWE subcategory: CWE-841 [2]

Description

As mentioned in Section 3.2, `Binopoly` has an essential `BrickFactory` contract to allow for tokenized materials and their assembly. This contract supports the `levelUp()` operation to upgrade the level by burning the required level-specific `updateMaterialNeeded`.

To elaborate, we show below the implementation of this `levelUp()` function. It comes to our attention the current implementation simply resets `_currentCollectedMaterial` to zero after the level upgrade (line 274). A possibly better approach to the user will be to deduct the `materialNeeded` from `_currentCollectedMaterial`, i.e., `_currentCollectedMaterial -= materialNeeded`!

```
260     // call "setApproveForAll" method to set this contract as the operator of the
            _msgSender()
261     // BEFORE calling this method
262     function levelUp(uint256 amount) public {
263         require(currentLevel < 6, "current factory level can not exceed 6");
264 
265         // read material needed for current level
266         FactoryAttributes storage thisLevelAttributes = _currentAttributes[currentLevel
                ];
267         uint256 materialNeeded = thisLevelAttributes.updateMaterialNeeded;
268         // burn injected materials, and upgrate collected amount
269         materialsAddress.burn(_msgSender(), basicMaterialId, amount);
270         _currentCollectedMaterial = _currentCollectedMaterial.add(amount);
271 
272         if (_currentCollectedMaterial >= materialNeeded) {
```

```
                _currentCollectedMaterial = 0;
                currentLevel = currentLevel.add(1);

                emit LevelUp(currentLevel);
            }
    }
```

Listing 3.4: `BrickFactory::levelUp()`

Recommendation Revise the above `levelUp()` function to properly update `currentCollectedMaterial`
.

Status This issue has been resolved as the number will be reset as the level is increased.

3.4 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Time and State [4]
- CWE subcategory: CWE-663 [1]

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` [9] exploit, and the recent `Uniswap/Lendf.Me` hack [8].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `BrickFactory` as an example, the `startProcess()` function (see the code snippet below) is provided to externally call a token contract 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 295) starts before effecting the update on the internal state (lines 301-303), 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.

```
    function startProcess(uint256 lineId, uint256 amount) public {
        require(lineId != 0, "can not process line #0");
        require(amount != 0, "can not process 0 amount");
        FactoryAttributes storage thisLevelAttributes = _currentAttributes[currentLevel
            ];
        require(thisLevelAttributes.productionLineLimit >= lineId, "exceed production
            line limit");
        require(thisLevelAttributes.processLimit >= amount, "exceed process limit");
        // check if this lineId is available
        require(!_linesUsage.contains(lineId), "this line is used by others now, try
            later");

        // pay Bino fee, and burn basic materials to start upgrating
        uint256 totalBinoFee = amount.mul(thisLevelAttributes.singleProcessPrice);  //
            in Bino's decimals: 1e18
        binoAddress.safeTransferFrom(_msgSender(), address(this), totalBinoFee);
        materialsAddress.burn(_msgSender(), basicMaterialId, amount);

        // set new time, amounts, and process user
        uint256 totalTime = amount.mul(thisLevelAttributes.singleProcessPeriod);
        uint256 finishTime = block.timestamp.add(totalTime);
        _linesUsage.set(lineId, _msgSender());
        _completeTime[lineId] = finishTime;
        _processAmounts[lineId] = amount;

        emit StartProcess(lineId, _msgSender(), finishTime, amount);
    }
```

Listing 3.5: `BrickFactory::startProcess()`

Note that other routines share the same issue, including `BinoDistributionLaw::claimBinoRewards()`, `BinoMarket::withdrawStakingProofSelling()/buyHouse()/buyStakingProof()/offerMaterialForSale()/withdrawMaterialSelling()`, and `ClayFarm::deposit()/withdraw()/depositCAKE()/withdrawCAKE()`.

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

Status This issue has been fixed in the following commit: f881054.

3.5 Proper pendingRewards() Calculation in BinoDistributionLaw

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: BinoDistributionLaw
- Category: Business Logic [3]
- CWE subcategory: CWE-841 [2]

Description

Binopoly has built-in incentive mechanisms to engage protocol users. While reviewing a specific BinoFarm pool, we notice the BinoDistributionLaw approach to compute pendingRewards() may return incorrect rewards.

To elaborate, we show below this function. Notice that the protocol is designed to reset the user's inject point to 0 for the new round and the new round is determined when the user's _lastTimeInject[account][lid] is at least CONTEST_DURATION + CLAIM_DURATION earlier. However, the following implementation shows it is determined as block.timestamp > _lastTimeInject[account][lid].add(CONTEST_DURATION) (line 146), which needs to be revised as block.timestamp > _lastTimeInject[account][lid].add(CONTEST_DURATION).add(CLAIM_DURATION).

```
140     // in Bino decimal, 1e18
141     function pendingRewards(address account, uint256 lid) public view returns (uint256)
            {
142         require(lid > 0 && lid <=7, "land id is out of range of [1, 7]");
143         if(!isClaimTime) {
144             return 0;
145         }
146         if(block.timestamp > _lastTimeInject[account][lid].add(CONTEST_DURATION)) {
147             return 0;
148         }
149 
150         uint256 thisRank = 0;
151         for (uint256 i = 0; i < rankForThisRound.length; ++i) {
152             if (lid == rankForThisRound[i]) {
153                 thisRank = i.add(1);    // from 1 to 3
154                 break;
155             }
156         }
157         if (thisRank == 0) {
158             return 0;
159         }
160 
161         uint256 rankShare;
162         if (thisRank == 1) {
```

```
            rankShare = _totalRewardBalance.mul(152).div(TOTAL_ALLO_POINTS);  // 1e18
        } else if (thisRank == 2) {
            rankShare = _totalRewardBalance.mul(69).div(TOTAL_ALLO_POINTS);  // 1e18
        } else {
            rankShare = _totalRewardBalance.mul(25).div(TOTAL_ALLO_POINTS);  // 1e18
        }

        return _injectedPointsOf[account][lid].mul(rankShare).div(_totalInjectedPointsOf
            [lid]);
    }
```

Listing 3.6: `BinoDistributionLaw::pendingRewards()`

The `claimBinoRewards()` function in the same contract shares the same issue.

Recommendation Revise the above-mentioned functions to properly determine whether the new round is entered.

Status This issue has been fixed in the following commit: f881054.

3.6 Timely massUpdatePools During Pool Weight Changes

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `ClayFarm`, `LicenseFarm`
- Category: Business Logic [3]
- CWE subcategory: CWE-841 [2]

Description

As mentioned earlier, `Binopoly` provides incentive mechanisms that reward the staking of supported assets. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of LP tokens in the reward pool.

The reward pools can be dynamically added via `add()` and the weights of supported pools can be adjusted via `set()`. When analyzing the pool weight update routine `set()`, we notice the need of timely invoking `massUpdatePools()` to update the reward distribution before the new pool weight becomes effective.

```
    // Update the given pool's MATERIAL allocation point. Can only be called by the
        owner.
    function set(
        uint256 _pid,
        uint256 _allocPoint,
        bool _withUpdate
```

```
    ) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(
            _allocPoint
        );
        poolInfo[_pid].allocPoint = _allocPoint;
    }
```

Listing 3.7: ClayFarm::set()

If the call to `massUpdatePools()` is not immediately invoked before updating the pool weights, certain situations may be crafted to create an unfair reward distribution. Moreover, a hidden pool without any weight can suddenly surface to claim unreasonable share of rewarded tokens. Fortunately, this interface is restricted to the owner (via the `onlyOwner` modifier), which greatly alleviates the concern. Note the same routine from the `LicenseFarm` contract shares the same issue.

Recommendation Timely invoke `massUpdatePools()` when any pool's weight has been updated. In fact, the third parameter (`_withUpdate`) to the `set()` routine can be simply ignored or removed.

```
    // Update the given pool's MATERIAL allocation point. Can only be called by the
        owner.
    function set(
        uint256 _pid,
        uint256 _allocPoint,
        bool _withUpdate
    ) public onlyOwner {
        massUpdatePools();
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(
            _allocPoint
        );
        poolInfo[_pid].allocPoint = _allocPoint;
    }
```

Listing 3.8: Revised ClayFarm::set()

Status This issue has been fixed in the following commit: f881054.

3.7 Duplicate Pool Detection and Prevention

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: ClayFarm
- Category: Business Logic [3]
- CWE subcategory: CWE-841 [2]

Description

Binopoly 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 (via a privileged function). 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 add(), 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 onlyOwner), it is still desirable to enforce it at the smart contract code level, eliminating the concern of wrong pool introduction from human omissions.

```
    // Add a new lp to the pool. Can only be called by the owner.
    // XXX DO NOT add the same LP token more than once. Rewards will be messed up if you
        do.
    function add(
        uint256 _allocPoint,
        IERC20 _lpToken,
        bool _withUpdate
    ) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        uint256 lastRewardBlock =
            block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(
            PoolInfo({
                lpToken: _lpToken,
                allocPoint: _allocPoint,
                lastRewardBlock: lastRewardBlock,
                accMaterialPerShare: 0
            })
```

```
        );
    }
```

Listing 3.9: `ClayFarm::add()`

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?");
        }
    }

    // Add a new lp to the pool. Can only be called by the owner.
    // XXX DO NOT add the same LP token more than once. Rewards will be messed up if you do.
    function add(
        uint256 _allocPoint,
        IERC20 _lpToken,
        bool _withUpdate
    ) public onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        checkPoolDuplicate(_lpToken);

        uint256 lastRewardBlock =
            block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint.add(_allocPoint);
        poolInfo.push(
            PoolInfo({
                lpToken: _lpToken,
                allocPoint: _allocPoint,
                lastRewardBlock: lastRewardBlock,
                accMaterialPerShare: 0
            })
        );
    }
```

Listing 3.10: Revised `ClayFarm::add()`

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 in the following commit: f881054.

3.8 Staking Incompatibility With Deflationary Tokens

- ID: PVE-008
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: ClayFarm
- Category: Business Logic [3]
- CWE subcategory: CWE-841 [2]

Description

In Binopoly, the ClayFarm contract is designed to take users' assets and deliver rewards depending on their share. In particular, one interface, i.e., deposit(), accepts asset transfer-in and records the depositor's balance. Another interface, i.e, withdraw(), allows the user to withdraw the asset with necessary bookkeeping under the hood. For the above two operations, i.e., deposit() and withdraw(), the contract using the safeTransfer()/safeTransferFrom() routines to transfer assets into or out of its pool. This routine works as expected with standard ERC20 tokens: namely the pool's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
    function deposit(uint256 _pid, uint256 _amount) public {
        require(_pid != 0, "can not stake CAKE in this pool");

        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pending =
                user.amount.mul(pool.accMaterialPerShare).div(1e30).sub(
                    user.rewardDebt
                );
            materials.mint(msg.sender, materialId, pending, "materialId minted!");
        }
        pool.lpToken.safeTransferFrom(
            address(msg.sender),
            address(this),
            _amount
        );
        user.amount = user.amount.add(_amount);
        user.rewardDebt = user.amount.mul(pool.accMaterialPerShare).div(1e30);
        emit Deposit(msg.sender, _pid, _amount);
    }
```

Listing 3.11: ClayFarm::deposit())

However, there exist other ERC20 tokens that may make certain customization to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every transfer.

As a result, this may not meet the assumption behind asset-transferring routines. In other words, the above operations, such as `deposit()` and `withdraw()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts. Apparently, these balance inconsistencies are damaging to accurate and precise portfolio management of the pool and affects protocol-wide operation and maintenance.

Specially, if we take a look at the `updatePool()` routine. This routine calculates `pool.accMaterialPerShare` via dividing `materialReward` by `lpSupply`, where the `lpSupply` is derived from `pool.lpToken.balanceOf(address(this))` (line 304). Because the balance inconsistencies of the pool, the `lpSupply` could be 1 `Wei` and thus may yield a huge `pool.accMaterialPerShare` as the final result, which dramatically inflates the pool's reward.

```
298     // Update reward variables of the given pool to be up-to-date.
299     function updatePool(uint256 _pid) public {
300         PoolInfo storage pool = poolInfo[_pid];
301         if (block.number <= pool.lastRewardBlock) {
302             return;
303         }
304         uint256 lpSupply = pool.lpToken.balanceOf(address(this));
305         if (_pid == 0) {
306             lpSupply = cakePoolBalance;
307         }
308         if (lpSupply == 0) {
309             pool.lastRewardBlock = block.number;
310             return;
311         }
312         uint256 materialReward =(block.number.sub(pool.lastRewardBlock))
313                 .mul(materialPerBlock).mul(pool.allocPoint).div(
314                 totalAllocPoint
315             );
316 
317         pool.accMaterialPerShare = pool.accMaterialPerShare.add(
318             materialReward.mul(1e30).div(lpSupply)
319         );
320         pool.lastRewardBlock = block.number;
321     }
```

Listing 3.12: `ClayFarm::updatePool()`

One mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in `safeTransfer()` or `safeTransferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the pool before and after the `safeTransfer()` or `safeTransferFrom()` is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into `Binopoly` for support. However, certain existing stable coins may exhibit control switches that can be dynamically exercised to convert into deflationary.

Recommendation Check the balance before and after the `safeTransfer()` or `safeTransferFrom()` call to ensure the book-keeping amount is accurate. An alternative solution is using non-deflationary tokens as collateral but some tokens (e.g., USDT) allow the admin to have the deflationary-like features kicked in later, which should be verified carefully.

Status This issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the design and implementation of `Binopoly`, which is a play-to-earn game to introduce the crypto finance concept to everyone, using a similar approach from the most popular financial educational board game in history. The current code base is clearly 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-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

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

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

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

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

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

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

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

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