SMART CONTRACT AUDIT REPORT

for

Light Year Game

Prepared By: Yiqun Chen

Hangzhou, China

January 24, 2022

Document Properties

Client	Light Year Game
Title	Smart Contract Audit Report
Target	Light Year Game
Version	1.0
Author	Jing Wang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Jing Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	January 24, 2022	Jing Wang	Final Release
1.0-rc	January 5, 2022	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 `Light Year Game` design document and related smart contract source code, 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 branch of `Light Year Game` 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 Light Year Game

`Light Year Game` is a decentralized space strategy game based on `Binance Smart Chain`. `Light Year Game` provides two game modes for two types of players: `DeFi` farmers and `P2E` players. `DeFi` farmers can maximize their yield farming return while `P2E` players can earn rewards based on their in-game performance. Players are free to mine natural resources, build starships, summon heroes, battle against other players, trade with each other in the marketplace, and join alliances or nations to fight side by side.

The basic information of `Light Year Game` is as follows:

Table 1.1: Basic Information of Light Year Game

Item	Description
Name	Light Year Game
Website	https://lightyear.game/home
Type	Ethereum and BSC Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	January 24, 2022

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

- https://github.com/LightYearGame/light-year-core (0f3b4ac)

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

- https://github.com/LightYearGame/light-year-core (0149f0b)

1.2 About PeckShield

PeckShield Inc. [15] 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 [14]:

- 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) [13], 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 `Light Year Game` 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 logic, 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	6	■■■■■■
Informational	1	■
Total	9	

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, 6 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Light Year Game Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Possible Overflow Prevention With SafeMath	Coding Practices	Fixed
PVE-002	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-003	Low	Improved Validation of Function Arguments	Coding Practices	Fixed
PVE-004	Low	Possible Sandwich/MEV Attacks For Reduced Returns	Time and State	Mitigated
PVE-005	Low	Accommodation of approve() Idiosyncrasies	Coding Practices	Fixed
PVE-006	Low	Incompatibility with Deflationary Tokens	Business Logic	Confirmed
PVE-007	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	Fixed
PVE-008	Medium	Weak Randomness When Creating New Hero And Ship	Business Logic	Mitigated
PVE-009	Low	Duplicate Pool Detection and Prevention	Coding Practices	Fixed

Beside the identified issues, we emphasize that for any user-facing applications and services, 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 should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Possible Overflow Prevention With SafeMath

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [9]
- CWE subcategory: CWE-1041 [1]

Description

`SafeMath` is a widely-used Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. While analyzing the `Light Year Game` implementation, we observe it can be improved by taking advantage of the improved security from `SafeMath`. In the following, we examine three cases.

The first case is the computation in `ClaimConfig::getClaimAmount()`: `64 * (2 ** research().levelMap(who_, 0))* 1e18` (line 21). The multiplication of `2 ** research().levelMap(who_, 0)` to `1e18` is not guarded against possible overflow. We should point out that this multiplication will not overflow in this particular usage scenario. However, it is always preferable to guarantee the overflow will always be detected and blocked.

```
20     function getClaimAmount(address who_) external override view returns(uint256) {
21         return 64 * (2 ** research().levelMap(who_, 0)) * 1e18;
22     }
```

Listing 3.1: `ClaimConfig::getClaimAmount()`

The second case is the computation in `BaseConfig::getCostArray()`: `100 * (2 ** level_)* 1e18` (lines 74-78). The multiplications are not guarded against possible overflow.

```
71     function getCostArray(uint256 itemIndex_, uint256 level_) external override view
           returns (uint256[] memory) {
72         uint256[] memory result = new uint256[](2);
73         if (itemIndex_ == 0) {
74             result[0] = 100 * (2 ** level_) * 1e18;
```

```
            result[1] = 100 * (2 ** level_) * 1e18;
        } else {
            result[0] = (level_ + 1) * 75 * 1e18;
            result[1] = (level_ + 1) * 85 * 1e18;
        }
        return result;
    }
```

Listing 3.2: `BaseConfig::getCostArray()`

The third case is the computation in `MiningConfig::getMultiplier()`: `(100 + base().levelMap(who_, 0))*(100 + base().levelMap(who_, assetIndex_ + 1))` (line 28). It is suggested to replace it with `(100.add(base().levelMap(who_, 0))).mul(100.add(base().levelMap(who_, assetIndex_ + 1)))`.

```
24      function getMultiplier(
25          address who_,
26          uint256 assetIndex_
27      ) external override view returns(uint256) {
28          return (100 + base().levelMap(who_, 0)) *
29              (100 + base().levelMap(who_, assetIndex_ + 1));
30      }
```

Listing 3.3: `MiningConfig::getMultiplier()`

Recommendation Make use of `SafeMath` in the above calculations to better mitigate possible overflows.

Status The issue has been fixed by this commit: `6ccc636`.

3.2 Trust Issue of Admin Keys

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [8]
- CWE subcategory: CWE-287 [3]

Description

In the `Light Year Game` protocol, there is a special administrative account, i.e., `owner`. This `owner` account plays a critical role in governing and regulating the protocol-wide operations (e.g., configure `operatorMap`). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged `owner` account and its related privileged accesses in current contract.

To elaborate, we show below the function provided to configure the `operatorMap` map, which stores the critical information about whether the `operator_` is able to transfer the funds directly without any allowance from the owner.

```
    function setOperator(address operator_) public onlyOwner {
        operatorMap[operator_] = true;
    }
```

Listing 3.4: `Registry::setOperator()`

```
    function operatorTransfer(address sender_, address recipient_, uint256 amount_)
        public onlyOperator {
        _transfer(sender_, recipient_, amount_);
    }
```

Listing 3.5: `CommodityERC20::operatorTransfer()`

We understand the need of the privileged functions for contract maintenance, but it is worrisome if the privileged `owner` 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 issue has been mitigated by this commit: `84ac4e9`.

3.3 Improved Validation of Function Arguments

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Ship`
- Category: Coding Practices [9]
- CWE subcategory: CWE-628 [4]

Description

In the `Ship` contract, the `buildShip()` function allows the user to use the their tokens to build new ship with selected `shipType_`. When the contract receives the tokens defined in `tokenArray`, it will mint a new ship and assign it to the user. To elaborate, we show below the related code snippet.

```
    function buildShip(uint8 shipType_) public {
        address[] memory tokenArray = shipConfig().getBuildTokenArray(shipType_);
```

```
        uint256[] memory costs = shipConfig().getBuildShipCost(shipType_);

        for (uint i = 0; i < tokenArray.length; i++) {
            ICommodityERC20(tokenArray[i]).operatorTransfer(_msgSender(), address(this),
                costs[i]);
            ICommodityERC20(tokenArray[i]).burn(costs[i]);
        }

        _mintShip(_msgSender(), shipType_);
    }
```

Listing 3.6: `Ship::buildShip()`

It comes to our attention that the `buildShip()` function has the inherent assumption on the same length of the given two arrays, i.e., `tokenArray` and `costs`. However, this is not enforced in the `buildShip()` function.

Recommendation Make the requirement of `tokenArray.length == costs.length` explicitly in the `buildShip()` function.

Status The issue has been fixed by this commit: `ec3574e`.

3.4 Possible Sandwich/MEV Attacks For Reduced Returns

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Staking`
- Category: Time and State [12]
- CWE subcategory: CWE-682 [6]

Description

The `Staking` contract has a helper routine, i.e., `_processReward()`, that is designed to process the reward tokens. It has a rather straightforward logic to swap `rewardToken` to `stableToken` by calling the `_swapIntoStableToken()` routine to actually perform the intended token swap.

```
    function _swapIntoStableToken(address fromToken_, uint256 fromAmount_) private
        returns(uint256) {
        address[] memory path = new address[](2);
        path[0] = address(fromToken_);
        path[1] = address(stableToken());
        uint256 deadline = now + (1 hours);

        IERC20(fromToken_).approve(registry.uniswapV2Router(), fromAmount_);
        uint256[] memory amounts = IUniswapV2Router02(registry.uniswapV2Router()).
            swapExactTokensForTokens(
            fromAmount_,
```

```
                0,
                path,
                address(this),  // TODO: stores stable token in a contract and buy back and burn.
                deadline);
            return amounts[1];
        }
```

Listing 3.7: `Staking::_swapIntoStableToken()`

To elaborate, we show above the `_swapIntoStableToken()` routine. We notice the token swap is routed to `registry.uniswapV2Router()` and the actual swap operation `swapExactTokensForTokens()` essentially does not specify any restriction (with `amountOutMin=0`) on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of yielding.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation to the above front-running attack to better protect the interests of farming users.

Status The issue has been confirmed by the teams. And the team clarifies that, the swap will be triggered as frequently as possible to make sure the swap amount is always small enough, to prevent sandwich attacks from happening.

3.5 Accommodation of approve() Idiosyncrasies

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Staking`
- Category: Coding Practices [9]
- CWE subcategory: CWE-1126 [2]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `approve()` routine and analyze possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
    /**
    * @dev Approve the passed address to spend the specified amount of tokens on behalf
        of msg.sender.
    * @param _spender The address which will spend the funds.
    * @param _value The amount of tokens to be spent.
    */
    function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

        // To change the approve amount you first have to reduce the addresses`
        //  allowance to zero by calling `approve(_spender, 0)` if it is not
        //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.8: USDT Token Contract

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. In the following, we use the `Staking::_swapIntoStableToken()` routine as an example. This routine is designed to swap `rewardToken` to `stableToken`. To accommodate the specific idiosyncrasy, for each `approve()` (line 143), there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

```
137    function _swapIntoStableToken(address fromToken_, uint256 fromAmount_) private
           returns(uint256) {
138        address[] memory path = new address[](2);
139        path[0] = address(fromToken_);
140        path[1] = address(stableToken());
141        uint256 deadline = now + (1 hours);

143        IERC20(fromToken_).approve(registry.uniswapV2Router(), fromAmount_);
144        uint256[] memory amounts = IUniswapV2Router02(registry.uniswapV2Router()).
               swapExactTokensForTokens(
145            fromAmount_,
146            0,
147            path,
148            address(this), // TODO: stores stable token in a contract and buy back and
                   burn.
```

```
                deadline);
        return amounts[1];
    }
```

Listing 3.9: AaveMarket::deposit()

Moreover, it is important to note that for certain non-compliant ERC20 tokens (e.g., USDT), the `transfer()` function does not have a return value. However, the `IERC20` interface has defined the `transfer()` interface with a `bool` return value. As a result, the call to `transfer()` may expect a return value. With the lack of return value of `USDT`'s `transfer()`, the call will be unfortunately reverted.

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`.

Status This issue has been fixed in this commit: `c905589`.

3.6 Incompatibility with Deflationary Tokens

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Staking`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

In the `Light year Game` protocol, the `Staking` contract is designed to take users' assets and deliver rewards depending on their deposit amount. In particular, one interface, i.e., `deposit0()`, accepts asset transfer-in and records the depositor's balance. Another interface, i.e, `withdraw0()`, allows the user to withdraw the asset. For the above two operations, i.e., `deposit0()` and `withdraw0()`, the contract makes the use of `safeTransferFrom()` or `safeTransfer()` 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 deposit0(uint256 pid_, uint256 amount_) external {
        PoolInfo storage pool = poolInfoArray[pid_];
```

```
        UserInfo storage user = userInfoMap[_msgSender()];
        UserPoolInfo storage userPool = userPoolInfoMap[_msgSender()][pid_];

        require(pool.token != address(0), "token is 0");
        require(pool.rewardToken != address(0), "reward token is 0");
        require(pool.token != pool.rewardToken, "Case not handled");

        uint256 balanceBefore = IERC20(pool.rewardToken).balanceOf(address(this));

        IERC20(pool.token).safeTransferFrom(_msgSender(), address(this), amount_);
        IERC20(pool.token).approve(address(pool.farm), amount_);
        pool.farm.deposit(pool.farmPid, amount_);

        uint256 balanceAfter = IERC20(pool.rewardToken).balanceOf(address(this));
        uint256 rewardAmount = balanceAfter.sub(balanceBefore);

        _processReward(pid_, rewardAmount);

        if (userPool.amount > 0) {
            uint256 pending = userPool.amount.mul(pool.accRewardPerShare).div(
                UNIT_PER_SHARE).sub(userPool.rewardDebt);
            user.rewardAmount = user.rewardAmount.add(pending);
        }

        pool.amount = pool.amount.add(amount_);
        userPool.amount = userPool.amount.add(amount_);
        userPool.rewardDebt = userPool.amount.mul(pool.accRewardPerShare).div(
            UNIT_PER_SHARE);
    }

    function withdraw0(uint256 pid_, uint256 amount_) external {
        PoolInfo storage pool = poolInfoArray[pid_];
        UserInfo storage user = userInfoMap[_msgSender()];
        UserPoolInfo storage userPool = userPoolInfoMap[_msgSender()][pid_];

        require(pool.token != address(0), "token is 0");
        require(pool.rewardToken != address(0), "reward token is 0");
        require(pool.token != pool.rewardToken, "Case not handled");

        uint256 balanceBefore = IERC20(pool.rewardToken).balanceOf(address(this));

        pool.farm.withdraw(pool.farmPid, amount_);
        IERC20(pool.token).safeTransfer(_msgSender(), amount_);

        uint256 balanceAfter = IERC20(pool.rewardToken).balanceOf(address(this));
        uint256 rewardAmount = balanceAfter.sub(balanceBefore);

        _processReward(pid_, rewardAmount);

        if (userPool.amount > 0) {
            uint256 pending = userPool.amount.mul(pool.accRewardPerShare).div(
                UNIT_PER_SHARE).sub(userPool.rewardDebt);
```

```
            user.rewardAmount = user.rewardAmount.add(pending);
        }

        pool.amount = pool.amount.sub(amount_);
        userPool.amount = userPool.amount.sub(amount_);
        userPool.rewardDebt = userPool.amount.mul(pool.accRewardPerShare).div(
            UNIT_PER_SHARE);
    }
```

Listing 3.10: `Staking::deposit0()/withdraw0()`

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 `deposit0()` and `withdraw0()`, 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.

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 `Staking` for support.

Recommendation Check the balance before and after the `safeTransfer()` or `safeTransferFrom()` call to ensure the book-keeping amount is accurate.

Status This issue has been confirmed. The team clarifies that they won't support deflationary tokens.

3.7 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Staking`
- Category: Time and State [11]
- CWE subcategory: CWE-663 [5]

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` [17] exploit, and the recent `Uniswap/Lendf.Me` hack [16].

We notice there are occasions where the `checks-effects-interactions` principle is violated. Using the `Staking` as an example, the `withdraw0()` 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`. For example, the interaction with the external contracts (line 235) start before effecting the update on internal states (lines 247 – 249), 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.

```
223     function withdraw0(uint256 pid_, uint256 amount_) external {
224         PoolInfo storage pool = poolInfoArray[pid_];
225         UserInfo storage user = userInfoMap[_msgSender()];
226         UserPoolInfo storage userPool = userPoolInfoMap[_msgSender()][pid_];
227 
228         require(pool.token != address(0), "token is 0");
229         require(pool.rewardToken != address(0), "reward token is 0");
230         require(pool.token != pool.rewardToken, "Case not handled");
231 
232         uint256 balanceBefore = IERC20(pool.rewardToken).balanceOf(address(this));
233 
234         pool.farm.withdraw(pool.farmPid, amount_);
235         IERC20(pool.token).safeTransfer(_msgSender(), amount_);
236 
237         uint256 balanceAfter = IERC20(pool.rewardToken).balanceOf(address(this));
238         uint256 rewardAmount = balanceAfter.sub(balanceBefore);
239 
```

```
        _processReward(pid_, rewardAmount);

        if (userPool.amount > 0) {
            uint256 pending = userPool.amount.mul(pool.accRewardPerShare).div(
                UNIT_PER_SHARE).sub(userPool.rewardDebt);
            user.rewardAmount = user.rewardAmount.add(pending);
        }

        pool.amount = pool.amount.sub(amount_);
        userPool.amount = userPool.amount.sub(amount_);
        userPool.rewardDebt = userPool.amount.mul(pool.accRewardPerShare).div(
            UNIT_PER_SHARE);
    }
```

Listing 3.11: `Staking::withdraw0()`

In the meantime, we should mention that the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`. However, it is important to take precautions to thwart possible `re-entrancy`. Note that another routine `deposit0()` shares the same issue.

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-interactions` principle and utilizing the necessary `nonReentrant` modifier to block possible `re-entrancy`.

Status This issue has been fixed in this commit: `376b1f6`.

3.8 Weak Randomness When Creating New Hero And Ship

- ID: PVE-008
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

In the `Light Year Game` contract, the `_createHero()` function allows the users to summon `heros` with `LC` tokens. The newly created `heros` starts at grade 1, and will be assigned with a random type resulting in different attributes, e.g., only `SSR` and above `heros` have special skills.

```
    function _createHero(bool advance_) private view returns (Info memory){
        uint8 heroType = uint8(_randomHeroType(advance_));
        Info memory info = Info(1, heroType);
        return info;
    }
```

Listing 3.12: `Hero::_createHero()`

However, randomness on Ethereum is an existing problem with no proper solution except using an oracle. As shown in the following code snippet, the `_random()` function uses the hash of the `gasprice`, difficulty, gaslimit, number, timestamp of the current block to generate the pseudo-random `random` number, which result the `type` from `HeroConfig::randomHeroType()`. If a bad actor uses a contract to trigger `Hero::_createHero()`, the `random` number could be easily derived. Therefore, the malicious contract could revert when the type of the `Hero` is not the one she need and always pick up a `SSSR Hero`, which totally breaks the design.

```
    function _randomHeroType(bool advance_) private view returns (uint256){
        uint256 random = _random(1e18);
        uint256 heroType = heroConfig().randomHeroType(advance_, random);
        return heroType;
    }
```

Listing 3.13: `Hero::_randomHeroType()`

```
    function randomHeroType(bool advance_, uint256 random_) public view override returns (uint256){
        uint256 r1 = random_ % 100;
        uint256 r2 = _random(random_, 12);
        if (!advance_) {
            if (r1 < 90) {
                return r2;
            } else if (r1 < 98) {
                return r2 + 12;
            } else {
                return r2 + 24;
            }
        } else {
            if (r1 < 80) {
                return r2 + 12;
            } else if (r1 < 98) {
                return r2 + 24;
            } else {
                return r2 + 36;
            }
        }
    }

    /**
     * random
     */
    function _random(uint256 seed_, uint256 randomSize_) private view returns (uint256){
        uint256 nonce = seed_;
        uint256 difficulty = block.difficulty;
        uint256 gaslimit = block.gaslimit;
        uint256 number = block.number;
        uint256 timestamp = block.timestamp;
        uint256 gasprice = tx.gasprice;
        uint256 random = uint256(keccak256(abi.encodePacked(nonce, difficulty, gaslimit, number, timestamp, gasprice))) % randomSize_;
```

```
        return random;
    }
```

Listing 3.14: `HeroConfig::randomHeroType()/_random()`

Note another routine `Ship::_createShip()` shares the same issue.

Recommendation Use an oracle to feed the random seed instead of using Blockchain data.

Status This issue has been mitigated in this commit: `20dd57a`.

3.9 Duplicate Pool Detection and Prevention

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Staking`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

The `Light Year` protocol has a `Staking` contract that 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. The rewards for stakers are proportional to their share of LP tokens in the pool.

In current implementation, 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. If the same `token_` of two staking pools are added and deposited into the same farming pool, the rewards from the farming pool will be messed up because the deposits are from two staking pools but the rewards will only be delivered to one `Staking` contract. 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.

```
    function addPool(
        IFarm farm_,
        uint256 farmPid_,
        address token_,
        address rewardToken_
    ) public onlyOwner {
        poolInfoArray.push(PoolInfo({
            farm: farm_,
            farmPid: farmPid_,
            token: token_,
```

```
            rewardToken: rewardToken_,
            amount: 0,
            accRewardPerShare: 0,
            lastRewardBlock: 0
        }));
    }
```

Listing 3.15: `Staking::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(address token_) public {
        uint256 length = poolInfoArray.length;
        for (uint256 i = 0; i < length; ++i) {
            require(poolInfoArray[i].token != token_, "add: token_ is already added to
                the pool");
        }
    }

    function addPool(
        IFarm farm_,
        uint256 farmPid_,
        address token_,
        address rewardToken_
    ) public onlyOwner {
        checkPoolDuplicate(token_);
        poolInfoArray.push(PoolInfo({
            farm: farm_,
            farmPid: farmPid_,
            token: token_,
            rewardToken: rewardToken_,
            amount: 0,
            accRewardPerShare: 0,
            lastRewardBlock: 0
        }));
    }
```

Listing 3.16: Revised `Staking::addPool()`

Status This issue has been fixed in this commit: `e261dcc`.

4 Conclusion

In this audit, we have analyzed the `Light Year Game` design and implementation. `Light Year Game` is a decentralized space strategy game based on `Binance Smart Chain`. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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