SMART CONTRACT AUDIT REPORT

for

LEGENDS NEVER DIE

Prepared By: Yiqun Chen

PeckShield

August 24, 2021

Document Properties

Client	Legends Never Die
Title	Smart Contract Audit Report
Target	Legends Never Die
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 24, 2021	Xiaotao Wu	Final Release
1.0-rc1	July 12, 2021	Xiaotao Wu	Release Candidate #1

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 `Legends Never Die` 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 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 Legends Never Die

The `Legends Never Die` project is a multi-dapp ecosystem that can be broken down into various courses, including `Hyper Legend`, `Referral`, `Pools`, `Bridge`, `Auction Lobby`, `Legend Vault`, `Liquidity Mining`, and `Legend DAO`.

The basic information of Legends Never Die is as follows:

Table 1.1: Basic Information of Legends Never Die

Item	Description
Name	Legends Never Die
Website	https://legendsneverdie.ae/
Type	BEP20 Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 24, 2021

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

- https://github.com/mygittab/CurrySwap-Food-Vault-and-Liquidity-Mining-contracts (5df033f)
- https://github.com/mygittab/CurrySwap-Auction-lobby-Contract (b70edc2)

And here are the commit IDs after fixes for the issues found in the audit have been checked in:

- https://github.com/mygittab/LegendsNeverDie-MasterVault-SmartContract (d402b54)
- https://github.com/mygittab/LegendsNeverDie-AuctionLobby-SmartContract (96621bc)

1.2 About PeckShield

PeckShield Inc. [10] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain 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 the OWASP Risk Rating Methodology [9]:

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

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

Table 1.3: The Full Audit Checklist

Category	Checklist Items
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 checklist of 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) [8], 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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

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 Logic	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.

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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `Legends Never Die` protocol. 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	2	■ ■
Medium	2	■ ■
Low	3	■ ■ ■
Informational	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 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 high-severity vulnerabilities, 2 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Legends Never Die Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Business Logic Error in _claimWeekly()	Business Logic	Fixed
PVE-002	High	Business Logic Error in _claimYearlyReward()	Business Logic	Fixed
PVE-003	Medium	Business Logic Error in getOfferingTokens()	Business Logic	Confirmed
PVE-004	Medium	Improved Handling of Corner Cases in _getPreviousBalance()	Coding Practices	Fixed
PVE-005	Informational	Improved Handling of Corner Cases in _add()	Coding Practices	Fixed
PVE-006	Low	Unfair Token Yearly Reward Mechanism	Business Logic	Confirmed
PVE-007	Low	Accommodation Of Possible Non-Compliant ERC20 Tokens	Coding Practices	Fixed
PVE-008	Low	Assumed Trust on Admin Keys	Security Features	Confirmed

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 Business Logic Error in _claimWeekly()

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `MasterVault`
- Category: Business Logics [7]
- CWE subcategory: CWE-841 [4]

Description

In Legends Never Die, the users can stake their `LND` tokens into `Legend vault` and will get `BNB-ETH` plus `BEP20` tokens as rewards. There are two kinds of rewards: weekly rewards and yearly rewards. Weekly rewards are rewards that can be claimed after each week while yearly rewards are rewards that can be claimed after the auction lobby is over. When reviewing the implementation of the `MasterVault` contract, we notice that the `claimWeekly()` function has a business logic error which could allow users to claim more rewards than they deserve.

In the following, we show the related external function `claimWeekly()` that is designed to allow the `staker` to claim the weekly rewards.

```
    function claimWeekly(uint256 _vid) external nonReentrant returns (uint256 returnAmount, uint256 rewardAmount) {
        // returnAmount = _claim(_vid, _msgSender());
        VaultInfo storage vault = vaultInfo[_vid];
        UserInfo storage user = userInfo[_vid][_msgSender()];

        require(((vault.start < block.number) && (vault.stop < block.number)), "Vault is not started or ended");
        require(!user.claimed, "Tokens already claimed");

        (returnAmount, rewardAmount) = _claimWeekly(vault, user, _vid, _msgSender());

        if (returnAmount > 0) {
            //send tokens to user
            offeringToken.safeTransfer(_msgSender(), returnAmount);
```

```
367         }
368
369         if (rewardAmount > 0) {
370             //send reward tokens to user
371             rewardToken.safeTransfer(_msgSender(), rewardAmount);
372         }
373     }
```

Listing 3.1: `MasterVault::claimWeekly()`

The internal function `_claimWeekly()` (line 362) calculates the amounts of the specified `_vid` (week) rewards for the `msg.sender`. However, if the `msg.sender` did not stake any `LND` tokens into the `Legend vault` on this specified `_vid`, the value of `userBalance` (lines 385) should take from the previous vault with `userBalance` greater than 0 instead of from the latest vault into which the `msg.sender` staked. The reason is that the `msg.sender` may stake some `LND` tokens into the `Legend vault` several weeks after this specified `_vid` (week) and then claim weekly rewards for this specified `_vid`. By doing so, the `msg.sender` can claim more rewards than deserved because the wrong `userBalance` is used to calculate the rewards share.

```
375     function _claimWeekly(
376         VaultInfo storage vault,
377         UserInfo storage user,
378         uint256 _vid,
379         address _recipient
380     ) internal returns (uint256 returnAmount, uint256 rewardAmount) {
381         // calculation how BNB ETH will be sent to user
382
383         uint256 userBalance = user.balance;
384         if (userBalance == 0) {
385             userBalance = userInfo[stakedInfoUser[_recipient].lastStakeIndex][_recipient
                    ].balance;
386         }
387
388         uint256 vaultTotalBalance = vault.totalBalanceStored;
389
390         if (vaultTotalBalance == 0) {
391             vaultTotalBalance = _getPreviousBalanceStored(_vid);
392         }
```

Listing 3.2: `MasterVault::_claimWeekly()`

Recommendation The value of `userBalance` (lines 385) should take from the previous vault with `userBalance` greater than 0 instead of from the latest vault which the `msg.sender` staked into.

Status The issue has been fixed by this commit: `1133386`.

3.2 Business Logic Error in _claimYearlyReward()

- ID: PVE-002
- Severity: High
- Likelihood: High
- Impact: High
- Target: MasterVault
- Category: Business Logics [7]
- CWE subcategory: CWE-841 [4]

Description

As mentioned in Section 3.1, besides weekly rewards, the Legends Never Die users can also claim yearly rewards after the auction lobby is over. When reviewing the implementation of the MasterVault contract, we notice that the _claimYearlyReward() function has a business logic error which may make users unable to get yearly rewards.

As shown in the following code snippets, the internal function _claimYearlyReward() calculates the amounts of the yearly rewards for msg.sender. However, if msg.sender has claimed the weekly rewards from the latest vault which the msg.sender staked into, the value of user.claimed (lines 571) would be true and no yearly rewards can be claimed from the MasterVault contract by this msg.sender.

```
563     function _claimYearlyReward() internal returns (uint256 returnAmount, uint256
            rewardAmount) {
564         VaultInfo memory vault = vaultInfo[yearlyRewardIndex];
565 
566 
567         StakedInfo storage userStakedInfo = stakedInfoUser[_msgSender()];
568 
569         UserInfo storage user = userInfo[userStakedInfo.lastStakeIndex][_msgSender()];
570 
571         if (vault.stop <= block.number && !userStakedInfo.yearleRewardClaimed && !user.
                claimed) {
572             uint256 userStakedInfolyStrength = yearlyRewardIndex - userStakedInfo.
                    startIndex + 1;
573             uint256 userAmountStrength = (user.balance * 50 * 1e16) / 1e18;
574             uint256 userTotalStrength = userStakedInfolyStrength * userAmountStrength;
575             uint256 cummulativeTotalStrength = ((((yearlyRewardIndex + 1) * totalAmount)
                     - totalWeight) * 50 * 1e16) / 1e18;
576 
577             returnAmount = (((vault.allTotalSupply * 50 * 1e16) / 1e18) *
                    userTotalStrength) / cummulativeTotalStrength;
578 
579             if (vault.allTotalReward > 0) {
580                 rewardAmount = (((vault.allTotalReward * 50 * 1e16) / 1e18) *
                        userTotalStrength) / cummulativeTotalStrength;
581             }
582 
583             userStakedInfo.yearleRewardClaimed = true;
584         }
```

```
585     }
```

Listing 3.3: `MasterVault::_claimYearlyReward()`

Recommendation Remove "`&& !user.claimed`" from the `if` statement (lines 571).

Status The issue has been fixed by this commit: `8178f60`.

3.3 Business Logic Error in getOfferingTokens()

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: Low
- Target: `MasterVault`
- Category: Business Logics [7]
- CWE subcategory: CWE-841 [4]

Description

As mentioned in Section 3.1, the `Legends Never Die` users can stake their `LND` tokens into `Legend vault` and will get `BNB-ETH` plus `BEP20` tokens as rewards. The `BNB-ETH` reward will come from the auction lobby contribution. When reviewing the implementation of the `MasterVault` contract, we notice that the `getOfferingTokens()` function has a business logic error which may lead users to get less weely/yearly rewards.

As shown in the following code snippets, the external function `getOfferingTokens` can only be called by the contract `owner` to get offering tokens (`BNB-ETH`) from auction lobby. However, if the contract `owner` calls this function more than one time to get offering tokens for a specified `_vid`, the calculations for `vault.totalSupply` (lines 681) and `vault.totalSupply` (lines 687) may not be correct. The `vault.totalSupply` value should be equal to the cumulative sum of the `supply` value returned by internally calling the `auctionLobby.getOfferingTokens()` function. The `vault.allTotalSupply` should be the sum of `prevVault.allTotalSupply` and `vault.totalSupply()`.

```
675     function getOfferingTokens(uint256 _vid, uint256 _aid) external onlyOwner returns (
            bool success, uint256 supply) {
676         VaultInfo storage vault = vaultInfo[_vid];
677
678         (success, supply) = auctionLobby.getOfferingTokens(_aid, perCentOfAuctionTokens)
            ;
679
680         if (success) {
681             vault.totalSupply = supply;
682
683             if (_vid == 0) {
684                 vault.allTotalSupply = supply;
685             } else {
```

```
                VaultInfo storage prevVault = vaultInfo[_vid - 1];
                vault.allTotalSupply = prevVault.allTotalSupply + supply;
            }
        }
    }
```

Listing 3.4: `MasterVault::getOfferingTokens()`

Recommendation Take into consideration the scenario where the contract `owner` may call the `getOfferingTokens()` function more than one time to get offering tokens for a specified `_vid`. An example revision is shown below:

```
    function getOfferingTokens(uint256 _vid, uint256 _aid) external onlyOwner returns (
        bool success, uint256 supply) {
        VaultInfo storage vault = vaultInfo[_vid];

        (success, supply) = auctionLobby.getOfferingTokens(_aid, perCentOfAuctionTokens)
            ;

        if (success) {
            vault.totalSupply += supply;

            if (_vid == 0) {
                vault.allTotalSupply += supply;
            } else {
                VaultInfo storage prevVault = vaultInfo[_vid - 1];
                vault.allTotalSupply = prevVault.allTotalSupply + vault.totalSupply;
            }
        }
    }
```

Listing 3.5: `MasterVault::getOfferingTokens()`

Status The issue has been confirmed.

3.4 Improved Handling of Corner Cases in _getPreviousBalance()

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: MasterVault
- Category: Coding Practices [6]
- CWE subcategory: CWE-561 [3]

Description

The MasterVault smart contract allows users to stake their LND tokens into Legend vault to get rewards. As shown in the following code snippets, the Legends Never Die users can call the external function stake() to stake their LND tokens into a specified vault. When a user stake his/her LND tokens into a specified _vid for the first time, the internal function _getPreviousBalance() (lines 242) is called to get the LND token total balance of all users from the previous vaults.

```
    function stake(uint256 _vid, uint256 _amount) external nonReentrant returns (uint256
        ) {
        //get parameters
        VaultInfo storage vault = vaultInfo[_vid];
        UserInfo storage user = userInfo[_vid][_msgSender()];
        StakedInfo storage userStakedInfo = stakedInfoUser[_msgSender()];

        //checking, vault is started
        require(((vault.start <= block.number) && (vault.stop >= block.number)), "vault
            is not started or ended");

        //transfer Legends Never Die(FV) or Legends Never Die-BNB (LM) to contract
        baseToken.safeTransferFrom(_msgSender(), address(this), _amount);

        if (!vault.finalized) {
            vault.totalBalance += _getPreviousBalance(_vid);
            vault.totalBalanceStored += _getPreviousBalance(_vid);
            vault.finalized = true;
        }
```

Listing 3.6: MasterVault::stake()

We show below the current _getPreviousBalance() implementation. If there are no users staked their LND tokens into these previous vaults, the i-- operation is executed in the case of i = 0 (lines 385).

Note that this does not result in an incorrect return value from _getPreviousBalance(), but does cause the function to revert unnecessarily when the above corner case occurs. Moreover, if there are no users staked their `LND` tokens into the first `vault` (`vid = 0`), the users will be unable to stake their `LND` tokens into the following `vaults` due to the revert.

```
    function _getPreviousBalance(uint256 _vid) internal view returns (uint256 balance) {
        if (_vid > 0) {
            for (uint256 i = _vid - 1; i >= 0; i--) {
                VaultInfo storage vault = vaultInfo[i];

                if (vault.totalBalance > 0) {
                    balance = vault.totalBalance;

                    break;
                }
            }
        }
    }
```

Listing 3.7: `MasterVault::_getPreviousBalance()`

Note a number of routines can be similarly improved, including `MasterVault::_getPreviousBalanceStored()`, and `MasterVault::_exitAll()`.

Recommendation Revise the above decrement operation for `i` to avoid the unnecessary function revert.

Status The issue has been fixed by this commit: `5af3eba`.

3.5 Improved Handling of Corner Cases in _add()

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `MasterVault`
- Category: Coding Practices [6]
- CWE subcategory: CWE-561 [3]

Description

The `MasterVault` smart contract allows the contract `owner` to add `vault` for the `Legends Never Die` users to stake their `LND` tokens into. When reviewing the implementation of the `MasterVault` contract, we notice that the `_add()` function, which will be called when the the contract `owner` need to add a new `vault`, can be improved by applying more rigorous sanity checks.

As shown in the following code snippets, certain corner case may lead to an undesirable consequence. Specifically, when a new `vault` is added, the current implementation requires `_start >= block.number` (lines 174). However, if `_autoStart` is enabled, a new value is assigned to `_start` (lines

181). In this case, it is possible that `(lastStop + 1 + blocksPerWeek())< block.number`. Thus the final effective value of `_start` will be smaller than `block.number`, which is undesirable.

```
169     function _add(
170         uint256 _start,
171         bool _autoStart,
172         uint256 _reward
173     ) internal {
174         require((_start >= block.number), "start and stop are not valid");
175 
176         if (activateReward) {
177             require(_reward > 0, "invalid reward value");
178         }
179         uint256 stop;
180         if (_autoStart && lastStop > 0) {
181             _start = lastStop + 1;
182             stop = _start + blocksPerWeek();
183         } else {
184             require(_start > lastStop, "start is not valid");
185             stop = _start + blocksPerWeek();
186         }
```

Listing 3.8: `MasterVault::_add()`

Recommendation Add necessary validation after lines 181 to ensure `_start` falls in an appropriate range.

Status The issue has been fixed by this commit: `d674199`.

3.6 Unfair Token Yearly Reward Mechanism

- ID: PVE-006
- Severity: Low
- Likelihood: High
- Impact: Low
- Target: `MasterVault`
- Category: Business Logics [7]
- CWE subcategory: CWE-841 [4]

Description

As mentioned in Section 3.1, the `Legends Never Die` users can claim yearly rewards after the auction lobby is over. According to the `Legends Never Die` design, there are two parameters to consider for yearly reward calculation: `Weekly Strength` (`ws`) is the number of weeks a user's `LND` staked amount is present in the contract and `amount strength` (`as`) is 50% of the `LND` amount a user staked in the contract. (And the total user strength is computed as `ts = ws * as`.) The amount of `BNB-ETH` a staker can claim at the end of the year : [(50% of (35% of the total `BNB-ETH` received in the whole year))

* (total strength aka ts of a user)] / (cumulative total strength aka ts of all users). The amount of BEP20 token a staker can claim at the end of the year : [(50% Total BEP20 token reserved for the year for Legend vault) * (total strength aka ts of a user)] / (cumulative total strength aka ts of all users).

When reviewing the implementation of the MasterVault contract, we notice that the current token yearly reward mechanism might be unfair for certain stakers. To elaborate, we show below a function _claimYearlyReward() that is used to calculate the amounts of the yearly rewards for msg.sender. Note the calculation of userStakedInfolyStrength (weekly strength of a staker) for a user only depends on the start staking week of this user (lines 572) and the calculation of the userAmountStrength (amount strength of a staker) for a user only depends on this user's latest LND token balance in the Legend vault (lines 573). For simplicity, we give an example to illustrate the unfairness of the current yearly reward mechanism. Suppose yearlyRewardIndex is set to 51. Suppose user A only stakes once and staked 100 LND tokens into the Legend vault on week 1. Suppose user B stakes twice, firstly staked 1 LND token into the Legend vault on week 1 and secondly staked 99 LND tokens into the Legend vault on week 52. When the auction lobby is over, user A and user B can claim exactly the same yearly reward. This may not be fair for user A.

```
563     function _claimYearlyReward() internal returns (uint256 returnAmount, uint256
            rewardAmount) {
564         VaultInfo memory vault = vaultInfo[yearlyRewardIndex];
565
566
567         StakedInfo storage userStakedInfo = stakedInfoUser[_msgSender()];
568
569         UserInfo storage user = userInfo[userStakedInfo.lastStakeIndex][_msgSender()];
570
571         if (vault.stop <= block.number && !userStakedInfo.yearleRewardClaimed && !user.
                claimed) {
572             uint256 userStakedInfolyStrength = yearlyRewardIndex - userStakedInfo.
                    startIndex + 1;
573             uint256 userAmountStrength = (user.balance * 50 * 1e16) / 1e18;
574             uint256 userTotalStrength = userStakedInfolyStrength * userAmountStrength;
575             uint256 cummulativeTotalStrength = ((((yearlyRewardIndex + 1) * totalAmount)
                     - totalWeight) * 50 * 1e16) / 1e18;
576
577             returnAmount = (((vault.allTotalSupply * 50 * 1e16) / 1e18) *
                    userTotalStrength) / cummulativeTotalStrength;
578
579             if (vault.allTotalReward > 0) {
580                 rewardAmount = (((vault.allTotalReward * 50 * 1e16) / 1e18) *
                        userTotalStrength) / cummulativeTotalStrength;
581             }
582
583             userStakedInfo.yearleRewardClaimed = true;
584         }
```

```
    }
```

Listing 3.9: `MasterVault::_claimYearlyReward()`

Recommendation Take into consideration of the durations of the `LND` tokens staked in each `vault` when calculating the total user strength for a staker.

Status The issue has been confirmed.

3.7 Accommodation Of Possible Non-Compliant ERC20 Tokens

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact:Medium
- Target: `MasterVault`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1109 [1]

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 `transfer()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
```

```
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.10: ZRX.sol

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 `transferFrom()` as well, i.e., `safeTransferFrom()`

In the following, we show the `_exitAll()` routine in the `MasterVault` contract. If the `ZRX` token is supported as the underlying `baseToken`, the unsafe version of `baseToken.transfer(_msgSender(), user.balance)` (lines 498) may return false in the `ZRX` token contract's `transfer()` implementation (but the `IERC20` interface expects a revert)! Thus, the contract has vulnerabilities against fake `transfer` attacks.

```
466     function _exitAll() internal {
467         //find first vault with balance from last
468         uint256 index = type(uint256).max;
469         for (uint256 i = vaultInfo.length - 1; i >= 0; i--) {
470             UserInfo storage user = userInfo[i][_msgSender()];
471 
472             if (user.balance > 0) {
473                 index = i;
474                 break;
475             }
476         }
477 
478         // if we find, we transfer balance from last
479         if (index != type(uint256).max) {
480             UserInfo storage user = userInfo[index][_msgSender()];
481             VaultInfo storage vault = vaultInfo[index];
482             StakedInfo storage userStakedInfo = stakedInfoUser[_msgSender()];
483 
484             //if finding vault is active, substruct user balance from this vault
485             if ((vault.start <= block.number) && (vault.stop >= block.number)) {
486                 vault.totalBalanceStored -= user.balance;
487             }
488 
489             vault.totalBalance -= user.balance;
490 
491             //update total amount
492             totalAmount -= user.balance;
493 
```

```
            //update weight, startIndex * all user balance
            totalWeight -= (userStakedInfo.startIndex * user.balance);

            //transfer from first finding vault with balance (cummulative balance)
            baseToken.transfer(_msgSender(), user.balance);

            userStakedInfo.staked = false;

            // nullify user balance from all vaults
            for (uint256 i = 0; i <= index; i++) {
                UserInfo storage user1 = userInfo[i][_msgSender()];
                if (user1.balance > 0) {
                    user1.balance = 0;
                }
            }
        }

        contributer[_msgSender()] = false;
    }
```

Listing 3.11: `MasterVault::_exitAll()`

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

Status The issue has been fixed by this commit: `8178f60`.

3.8 Assumed Trust on Admin Keys

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `MasterVault`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `Legends Never Die` smart contracts, there are some special accounts like `owner` and `admin` that play critical roles in governing and regulating the entire operation and maintenance. We examine closely the `MasterVault` and `AuctionLobby` contracts and identify trust issues on these accounts.

Firstly, we note that the `setYearlyRewardIndex()` function allows for the `owner` to set the `setYearlyRewardIndex` for the `Legend vault`. The `Legends Never Die` users can only claim the yearly rewards after the auction lobby is over which is determined by this `setYearlyRewardIndex`.

```
    function setYearlyRewardIndex(uint256 _yearlyRewardIndex) external onlyOwner {
        yearlyRewardIndex = _yearlyRewardIndex;
```

```
    }
```

Listing 3.12: `MasterVault::setYearlyRewardIndex()`

Secondly, we note that the `sweep()` function allows for the `owner` to sweep all the `LP` tokens to a specified account. Note the `AuctionLobby::sweep()` routine shares a similar issue.

```
    function sweep(address _recipient, address _token) external onlyOwner {
        IERC20Upgradeable(_token).safeTransfer(_recipient, IERC20Upgradeable(_token).
            balanceOf(address(this)));
    }
```

Listing 3.13: `MasterVault::sweep()`

Thirdly, we note that the `owner` is responsible for getting offering tokens from the auction lobby by calling the `getOfferingTokens()` function. These offering tokens are used as weekly and yearly rewards. Note the `AuctionLobby::getOfferingTokens()` routine shares a similar issue. The `admin role` can also get the offering tokens from the auction lobby.

```
    function getOfferingTokens(uint256 _vid, uint256 _aid) external onlyOwner returns (
        bool success, uint256 supply) {
        VaultInfo storage vault = vaultInfo[_vid];

        (success, supply) = auctionLobby.getOfferingTokens(_aid, perCentOfAuctionTokens)
            ;

        if (success) {
            vault.totalSupply = supply;

            if (_vid == 0) {
                vault.allTotalSupply = supply;
            } else {
                VaultInfo storage prevVault = vaultInfo[_vid - 1];
                vault.allTotalSupply = prevVault.allTotalSupply + supply;
            }
        }
    }
```

Listing 3.14: `MasterVault::getOfferingTokens()`


```
    function getOfferingTokens(uint256 _vid, uint256 _aid) external onlyOwner returns (
        bool success, uint256 supply) {
        VaultInfo storage vault = vaultInfo[_vid];

        (success, supply) = auctionLobby.getOfferingTokens(_aid, perCentOfAuctionTokens)
            ;

        if (success) {
            vault.totalSupply = supply;

            if (_vid == 0) {
                vault.allTotalSupply = supply;
```

```
            } else {
                VaultInfo storage prevVault = vaultInfo[_vid - 1];
                vault.allTotalSupply = prevVault.allTotalSupply + supply;
            }
        }
    }
```

Listing 3.15: AuctionLobby::getOfferingTokens()

Lastly, we note that in the updateBlocks() function, the owner has the right to update the _start and _stop values for vaults that are not ended yet. Note the AuctionLobby::updateBlocks() routine shares a similar issue.

```
    function updateBlocks(
        uint256 _vid,
        uint256 _start,
        uint256 _stop
    ) external onlyOwner {
        VaultInfo storage vault = vaultInfo[_vid];

        require(vault.stop >= block.number, "Auction is ended");

        if (vault.stop >= block.number) {
            vault.start = _start;
            vault.stop = _stop;
        } else {
            vault.stop = _stop;
        }
    }
```

Listing 3.16: MasterVault::updateBlocks()

We understand the need of the privileged functions for contract operation, but at the same time the extra power to the owner/admin may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among contract users.

Recommendation Make the list of extra privileges granted to the owner/admin explicit to Legends Never Die users.

Status The issue has been confirmed.

4 Conclusion

In this audit, we have analyzed the `Legends Never Die` design and implementation. The `Legends Never Die` project is a multi-dapp ecosystem that can be broken down into various courses, including `Hyper Legend`, `Referral`, `Pools`, `Bridge`, `Auction Lobby`, `Legend Vault`, `Liquidity Mining`, and `Legend DAO`. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

Moreover, we need to emphasize that `Solidity-based` 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-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

[3] MITRE. CWE-561: Dead Code. https://cwe.mitre.org/data/definitions/561.html.

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

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

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

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

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

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

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