SMART CONTRACT AUDIT REPORT

for

DSG

Prepared By: Yiqun Chen

PeckShield
November 24, 2021

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	November 24, 2021	Jing Wang	Final Release
1.0-rc	Nov 22, 2021	Jing Wang	Release Candidate

Contact

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

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

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the DSG protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About DSG

DSG is a DeFi project built on Binance Smart Chain (BSC). It provides a trading platform and NFT exchange market. The project includes Dinosaur Swap, NFT farming, and so on. As compared to traditional DEX projects, the tokenomics of Dinosaur Eggs is more innovative, with diverse token types to capture protocol income. The protocol has its own platform token, DSG, which can be minted into vDSG, a token representing a member's stake.

The basic information of the DSG protocol is as follows:

Table 1.1: Basic Information of The DSG Protocol

Item	Description
Name	DSG
Website	https://dsgmetaverse.com/
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 24, 2021

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

- https://github.com/Dinosaur-eggs/core (6f607f7)

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

- https://github.com/Dinosaur-eggs/core (04e16d3)

1.2 About PeckShield

PeckShield Inc. [16] 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 [15]:

- 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

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

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) [14], 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 DSG 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	1	■
Medium	2	■■
Low	10	■■■■■■■■■■
Informational	0	
Total	13	

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

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerabilities, 2 medium-severity vulnerabilities, and 10 low-severity vulnerabilities.

Table 2.1: Key DSG Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Timely massUpdatePools During Pool Weight Changes	Business Logic	Confirmed
PVE-002	Low	Incompatibility with Deflationary Tokens	Business Logics	Mitigated
PVE-003	Low	Suggested Adherence of Checks-Effects-Interaction Pattern	Time and State	Fixed
PVE-004	Low	Duplicate Pool Detection and Prevention In Erc20EarnNftPool	Business Logics	Fixed
PVE-005	Medium	Prohibited Contract Calls For computerSeed()	Business Logic	Fixed
PVE-006	Low	Fees Bypass With Direct ERC721 transferFrom()	Business Logic	Confirmed
PVE-007	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-008	Low	Possible Overflow in vDSGToken::_mint()	Coding Practices	Confirmed
PVE-009	Low	Safe-Version Replacement With safeTransfer() And safeTransferFrom()	Coding Practices	Fixed
PVE-010	Low	Possible Sandwich/MEV Attacks For Reduced Returns	Time and State	Confirmed
PVE-011	Low	Possible Price manipulation For Oracle::getLpTokenValue()	Time and State	Confirmed
PVE-012	High	Potential Fund Lockup For Contract Users	Business Logic	Fixed
PVE-013	Low	Improved Logic of LiquidityPoolOther::emergencyWithdraw()	Business Logic	Fixed

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 Timely massUpdatePools During Pool Weight Changes

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `LiquidityPool`
- Category: Business Logic [11]
- CWE subcategory: CWE-841 [7]

Description

The `DSG` protocol has a `LiquidityPool` contract that 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 reward token 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 3.1: `LiquidityPool::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 other routines `DepositPool::set()`, `TradingPool::set()`, `SinglePool::updateMultiplier()`, share 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 reward token 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 3.2: `LiquidityPool::set()`

Status The issue has been confirmed by the team. And the team clarifies that they would like to keep the `_withUpdate` in order to prevent updating too many pools to cause the `massUpdatePools()` routine unstable.

3.2 Incompatibility with Deflationary Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `DepositPool`
- Category: Business Logics [11]
- CWE subcategory: CWE-841 [7]

Description

In the `DSG` protocol, the `DepositPool` 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 `safeTransferFrom()` or `safeTransfer()` routine 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.

```
    // Deposit LP tokens to MasterChef for CAKE allocation.
    function deposit(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pendingAmount = user.amount.mul(pool.accRewardPerShare).div(1e12).
                sub(user.rewardDebt);
            if (pendingAmount > 0) {
                safeRewardTokenTransfer(msg.sender, pendingAmount);
                user.accRewardAmount = user.accRewardAmount.add(pendingAmount);
                pool.allocRewardAmount = pool.allocRewardAmount.sub(pendingAmount);
            }
        }
        if (_amount > 0) {
            ERC20(pool.token).safeTransferFrom(msg.sender, address(this), _amount);
            user.amount = user.amount.add(_amount);
            pool.totalAmount = pool.totalAmount.add(_amount);
        }
        user.rewardDebt = user.amount.mul(pool.accRewardPerShare).div(1e12);
        emit Deposit(msg.sender, _pid, _amount);
    }

    function withdraw(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][tx.origin];
        require(user.amount >= _amount, "DepositPool: withdraw: not good");
        updatePool(_pid);
        uint256 pendingAmount = user.amount.mul(pool.accRewardPerShare).div(1e12).sub(
            user.rewardDebt);
        if (pendingAmount > 0) {
            safeRewardTokenTransfer(tx.origin, pendingAmount);
            user.accRewardAmount = user.accRewardAmount.add(pendingAmount);
            pool.allocRewardAmount = pool.allocRewardAmount.sub(pendingAmount);
        }
        if (_amount > 0) {
            user.amount = user.amount.sub(_amount);
            pool.totalAmount = pool.totalAmount.sub(_amount);
            ERC20(pool.token).safeTransfer(tx.origin, _amount);
        }
        user.rewardDebt = user.amount.mul(pool.accRewardPerShare).div(1e12);
        emit Withdraw(tx.origin, _pid, _amount);
    }
```

Listing 3.3: `DepositPool::deposit()and DepositPool::withdraw()`

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()` or `transferFrom()`. 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.accRewardPerShare` via dividing `tokenReward` by `tokenSupply`, where the `tokenSupply` is derived from `balanceOf(address(this))` (line 177). Because the balance inconsistencies of the pool, the `tokenSupply` could be 1 `Wei` and thus may give a big `pool.accRewardPerShare` as the final result, which dramatically inflates the pool's reward.

```
170     // Update reward variables of the given pool to be up-to-date.
171     function updatePool(uint256 _pid) public {
172         PoolInfo storage pool = poolInfo[_pid];
173         if (block.number <= pool.lastRewardBlock) {
174             return;
175         }
176
177         uint256 tokenSupply = ERC20(pool.token).balanceOf(address(this));
178         if (tokenSupply == 0) {
179             pool.lastRewardBlock = block.number;
180             return;
181         }
182
183         uint256 blockReward = getRewardTokenBlockReward(pool.lastRewardBlock);
184
185         if (blockReward <= 0) {
186             return;
187         }
188
189         uint256 tokenReward = blockReward.mul(pool.allocPoint).div(totalAllocPoint);
190
191         bool minRet = rewardToken.mint(address(this), tokenReward);
192         if (minRet) {
193             pool.accRewardPerShare = pool.accRewardPerShare.add(tokenReward.mul(1e12).
                    div(tokenSupply));
194             pool.allocRewardAmount = pool.allocRewardAmount.add(tokenReward);
195             pool.accRewardAmount = pool.accRewardAmount.add(tokenReward);
196         }
197         pool.lastRewardBlock = block.number;
198     }
```

Listing 3.4: `DepositPool::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 `DepositPool` 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.

Status The issue has been confirmed by the team. And the team mitigated this issue by using `pool.totalAmount` to calculate `tokenSupply` instead of `balanceOf(address(this)` by the following commit `6b046e6`.

3.3 Suggested Adherence of Checks-Effects-Interaction Pattern

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `SinglePool`
- Category: Time and State [12]
- 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` [18] exploit, and the recent `Uniswap/Lendf.Me` hack [17].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `SinglePool` contract as an example, the `emergencyWithdraw()` 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 204) starts before effecting the update on internal states (lines 210−211), 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.

```
// Withdraw without caring about rewards. EMERGENCY ONLY.
function emergencyWithdraw() public {
    PoolInfo storage pool = poolInfo[0];
    UserInfo storage user = userInfo[msg.sender];
    pool.lpToken.safeTransfer(address(msg.sender), user.amount);
    if(totalDeposit >= user.amount) {
        totalDeposit = totalDeposit.sub(user.amount);
    } else {
        totalDeposit = 0;
    }
    user.amount = 0;
    user.rewardDebt = 0;
    emit EmergencyWithdraw(msg.sender, user.amount);
}
```

Listing 3.5: `SinglePool::emergencyUnstake()`

Note that other routines `SinglePool::deposit()` and `SinglePool::withdraw()` share the same issue.

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

Status The issue has been fixed by these commits: `238d29d` and `f642a15`.

3.4 Duplicate Pool Detection and Prevention In Erc20EarnNftPool

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Erc20EarnNftPool`
- Category: Business Logic [11]
- CWE subcategory: CWE-841 [7]

Description

The `Erc20EarnNftPool` protocol provides incentive mechanisms that reward the staking of supported assets with certain `NFT`. The rewards are carried out by designating a number of staking pools into which supported assets can be staked.

In current implementation, there are a number of concurrent pools and 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 `addPool()`, whose code logic is shown below. It turns out it did not perform necessary sanity checks in preventing a new pool but with a duplicate token from being added. Though it is a privileged interface (protected with the modifier `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(address _tokenAddress, uint256 _stakeAmount, uint256 _stakeTime,
        address _nftAddress, bool isLp) external onlyOwner {
        require(_tokenAddress.isContract(), "stake token address should be smart
            contract address");
        require(_nftAddress.isContract(), "NFT address should be smart contract address"
            );

        uint256[] memory tokenIds;

        if(isLp) {
            require(ISwapPair(_tokenAddress).token0() != address(0), "not lp");
        }

        pool.push(Pool({
            tokenAddress: _tokenAddress,
            isLpToken: isLp,
            stakeAmount: _stakeAmount,
            stakeTime: _stakeTime,
            nftAddress: _nftAddress,
            nftTokenIds: tokenIds,
            nftLeft: 0
        }));

        emit AddPoolEvent(_tokenAddress, _stakeAmount, _stakeTime, _nftAddress);
    }
```

Listing 3.6: `Erc20EarnNftPool::addPool()`

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

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

    function addPool(address _tokenAddress, uint256 _stakeAmount, uint256 _stakeTime,
        address _nftAddress, bool isLp) external onlyOwner {
        require(_tokenAddress.isContract(), "stake token address should be smart
            contract address");
        require(_nftAddress.isContract(), "NFT address should be smart contract address"
            );

```

```
        uint256[] memory tokenIds;

        if(isLp) {
            require(ISwapPair(_tokenAddress).token0() != address(0), "not lp");
        }

        checkPoolDuplicate(_tokenAddress);

        pool.push(Pool({
            tokenAddress: _tokenAddress,
            isLpToken: isLp,
            stakeAmount: _stakeAmount,
            stakeTime: _stakeTime,
            nftAddress: _nftAddress,
            nftTokenIds: tokenIds,
            nftLeft: 0
        }));

        emit AddPoolEvent(_tokenAddress, _stakeAmount, _stakeTime, _nftAddress);
    }
```

Listing 3.7: Revised `Erc20EarnNftPool::addPool()`

We point out that if a new pool with a duplicate LP token can be added, it will likely cause a havoc in the calculation of pool `LP` token balance.

Status The issue has been fixed by this commit: `40e6716`.

3.5 Prohibited Contract Calls For computerSeed()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `DsgNft`
- Category: Business Logic [11]
- CWE subcategory: CWE-841 [7]

Description

In the `DsgNft` contract, the `upgradeNft()` function calls the `Random.computerSeed()` routine to produce a random seed for `randomPower()` calculation.

```
    function upgradeNft(uint256 nftId, uint256 materialNftId) public override
        nonReentrant whenNotPaused
    {
        require(canUpgrade, "CANT UPGRADE");
        LibPart.NftInfo memory nft = getNft(nftId);
        LibPart.NftInfo memory materialNft = getNft(materialNftId);

```

```
        require(nft.level == materialNft.level, "The level must be the same");
        require(nft.level < maxLevel, "Has reached the max level");

        burn(nftId);
        burn(materialNftId);

        uint256 newLevel = nft.level + 1;
        uint256 fee = getUpgradeFee(newLevel);
        if (fee > 0) {
            _token.safeTransferFrom(_msgSender(), _feeWallet, fee);
        }

        uint256 seed = Random.computerSeed()/23;

        uint256 newId = _doMint(_msgSender(), nft.name, newLevel, randomPower(newLevel,
            seed), nft.res, nft.author);

        emit Upgraded(nftId, materialNftId, newId, newLevel, block.timestamp);
    }
```

Listing 3.8: `DsgNft::upgradeNft()`

However, the randomness on the `Ethereum` blockchain is an existing problem with no proper solution except using an oracle. As shown in the following code snippet, the `computerSeed()` function uses the hash of the block timestamp, `coinbase` of the current block, the `gaslimit`, and the block number to generate the pseudo-random `seed`.

If a bad actor uses a contract to trigger this function, the `seed` could be easily derived. Therefore, the malicious contract could revert when the `randomPower` is not the one it needs and always picks up a highest power to process, which breaks the randomness of the `NFT` upgrade.

```
    function computerSeed() internal view returns (uint256) {
        uint256 seed =
        uint256(
            keccak256(
                abi.encodePacked(
                    (block.timestamp)
                    .add(block.difficulty)
                    .add(
                        (
                        uint256(
                            keccak256(abi.encodePacked(block.coinbase))
                        )
                        ) / (block.timestamp)
                    )
                    .add(block.gaslimit)
                    .add(
                        (uint256(keccak256(abi.encodePacked(msg.sender)))) /
                        (block.timestamp)
                    )
                    .add(block.number)
                )
```

```
            )
        );
        return seed;
    }
```

Listing 3.9: `Random::computerSeed()`

Recommendation Use an oracle to feed the random seed instead of using blockchain data and also prohibit calling from contract to the `computerSeed()` routine.

Status The issue has been fixed by this commit: `dd6d19b`.

3.6 Fees Bypass With Direct ERC721 transferFrom()

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `NFTMarket`
- Category: Business Logic [11]
- CWE subcategory: CWE-841 [7]

Description

Each `NFT` supported in the `NFTMarket` protocol is represented as an `ERC721`-based NFT token, which naturally has the standard implementation, e.g., `transferFrom()/safeTransferFrom()`. By design, each piece of `NFT` for sale in `NFTMarket` will be listed on `_salesObjects`. Once sold, it will be transferred from the protocol to the buyer and the necessary `fees` are collected.

```
function buy(uint index)
    public
    nonReentrant
    mustNotSellingOut(index)
    checkTime(index)
    payable
{
    SalesObject storage obj = _salesObjects[index];
    require(obj.status == 0, "bad status");

    uint256 price = this.getSalesPrice(index);
    obj.status = 1;

    uint256 tipsFee = price.mul(_tipsFeeRate).div(_baseRate);
    uint256 purchase = price.sub(tipsFee);

    address currencyAddr = _saleOnCurrency[obj.id];
    if (currencyAddr == address(0)) {
        currencyAddr = TransferHelper.getETH();
    }
```

```
    uint256 royaltiesAmount;
    if(obj.nft.supportsInterface(bytes4(keccak256('getRoyalties(uint256)')))
        && _disabledRoyalties[address(obj.nft)] == false) {

        LibPart.Part[] memory fees = Royalties(address(obj.nft)).getRoyalties(obj.
            tokenId);
        for(uint i = 0; i < fees.length; i++) {
            uint256 feeValue = price.mul(fees[i].value).div(10000);
            if (purchase > feeValue) {
                purchase = purchase.sub(feeValue);
            } else {
                feeValue = purchase;
                purchase = 0;
            }
            if (feeValue != 0) {
                royaltiesAmount = royaltiesAmount.add(feeValue);
                if(TransferHelper.isETH(currencyAddr)) {
                    TransferHelper.safeTransferETH(fees[i].account, feeValue);
                } else {
                    IERC20(currencyAddr).safeTransferFrom(msg.sender, fees[i].account,
                        feeValue);
                }
            }
        }
    }

    if (TransferHelper.isETH(currencyAddr)) {
        require (msg.value >= this.getSalesPrice(index), "your price is too low");
        uint256 returnBack = msg.value.sub(price);
        if(returnBack > 0) {
            payable(msg.sender).transfer(returnBack);
        }
        if(tipsFee > 0) {
            IWOKT(WETH).deposit{value: tipsFee}();
            IWOKT(WETH).transfer(_tipsFeeWallet, tipsFee);
        }
        obj.seller.transfer(purchase);
    } else {
        IERC20(currencyAddr).safeTransferFrom(msg.sender, _tipsFeeWallet, tipsFee);
        IERC20(currencyAddr).safeTransferFrom(msg.sender, obj.seller, purchase);
    }

    obj.nft.safeTransferFrom(address(this), msg.sender, obj.tokenId);

    obj.buyer = payable(msg.sender);
    obj.finalPrice = price;

    // fire event
    emit eveSales(index, obj.tokenId, msg.sender, currencyAddr, price, tipsFee,
        royaltiesAmount, block.timestamp);
```

```
}
```

Listing 3.10: `NFTMarket::buy()`

To elaborate, we show above the `buy()` routine. This routine is used by the buyer to buy a `NFT` with proper `tipsFee` and `feeValue` payment. It comes to our attention that instead of using the build-in functions to trade `NFT`, it is possible for the seller and the buyer to directly negotiate a price, without paying the `fees`. The `NFT` can then be delivered by the seller directly calling `transferFrom()/safeTransferFrom()` with the buyer as the recipient.

Recommendation Implement a locking mechanism so that any `NFT`, needs to be locked in the `NFTMarket` contract in order to be available for public auction.

Status The issue has been confirmed. The team clarifies that they do not override the `transferFrom()/safeTransferFrom()` method to collect fee and users are free of transferring between each other.

3.7 Trust Issue of Admin Keys

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [8]
- CWE subcategory: CWE-287 [3]

Description

In the `DSG` protocol, there is a special administrative account, i.e., `owner`. This `owner` account plays a critical role in governing and regulating the system-wide operations (e.g., minting tokens , setting various parameters, etc.). 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 account and their related privileged accesses in current contracts.

To elaborate, we show below the `mint()` functions in the `DSGToken` contract, which allows the `Minter` to add tokens into circulation and the recipient can be directly provided when the mint operation takes place.

```
    function mint(address _to, uint256 _amount) public onlyMinter returns (bool) {
        uint256 teamAmount;
        if (teamWallet != address(0) && teamRate > 0) {
            teamAmount = _amount.mul(teamRate).div(10000);
        }

        _mint(_to, _amount);
```

```
        if (teamAmount > 0) {
            _mint(teamWallet, teamAmount);
        }
        return true;
    }
```

Listing 3.11: `DSGToken::mint()`

Our on-chain analysis shows that the `DSG` protocol used below mechanism to reward stakers with `DSGToken`:

- The `DSGToken` has a `mint()` function that allows its minters to mint new tokens.
- The `LiquidityPool`/`TradingPool`/`vDSGToken` contracts are the `DSGToken`'s owner and its internal logic rewards stakers with new `DSGTokens` minted.
- The `TimeLock` is the `DSGToken`'s owner and is allowed to pass transactions to add new minters with a configured delay.

The owner of the `TimeLock` contract is an `EOA` account, `0x8e8c01e78f15912c815407117893cf0226ca4f88`, which is owned by the `Dinosaur Eggs: Deployer`. While the current extra power to the owner is a counter-party risk to current contract users, the `timelock`-based administration greatly alleviates this concern, though a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

Also, the owners of other contracts in the `DSG` protocol are plain EOAs. Some owners take the important responsibility to withdraw all funds from the contract in emergency situations.

```
    function emergencyWithdraw() public onlyOwner {
        uint256 dsgBalance = IERC20(_dsgToken).balanceOf(address(this));
        IERC20(_dsgToken).safeTransfer(owner(), dsgBalance);
    }
```

Listing 3.12: `vDSGToken::emergencyWithdraw()`

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 confirmed by the team. The team clarifies that the `TimeLock` contract is currently managed by the team, and they will gradually switch to DAO in the future. Also the other `owners` will be transferred to the time lock after adjusting the reward coefficient.

3.8 Possible Overflow in vDSGToken::_mint()

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `vDSGToken`
- Category: Coding Practices [10]
- CWE subcategory: CWE-1041 [1]

Description

The `vDSG` token is a membership credential as well as a value capture tool of the `DSG` protocol. Users could pledge 100 `DSG` to mint one `vDSG` by calling the `mint()` routine. When the `vDSGToken` contract receives the funds, it will update the `StakingPower` of the user. However, our analysis shows that when a large amount of `DSG` tokens are transferred into the contract at one time, the calculation of `StakingPower` may overflow.

To elaborate, we show below the implementation code of the `_mint()` function:

```
397     function _mint(UserInfo storage to, uint256 stakingPower) internal {
398         require(stakingPower <= uint128(-1), "OVERFLOW");
399         UserInfo storage superior = userInfo[to.superior];
400         uint256 superiorIncreSP = DecimalMath.mulFloor(stakingPower, _superiorRatio);
401         uint256 superiorIncreCredit = DecimalMath.mulFloor(superiorIncreSP, alpha);

403         to.stakingPower = uint128(uint256(to.stakingPower).add(stakingPower));
404         to.superiorSP = uint128(uint256(to.superiorSP).add(superiorIncreSP));

406         superior.stakingPower = uint128(uint256(superior.stakingPower).add(
                superiorIncreSP));
407         superior.credit = uint128(uint256(superior.credit).add(superiorIncreCredit));

409         _totalStakingPower = _totalStakingPower.add(stakingPower).add(superiorIncreSP);
410     }
```

Listing 3.13: `vDSGToken::_mint()`

From the above code, we notice that in the computation of `to.stakingPower`: `uint128(uint256(to.stakingPower).add(stakingPower))` (line 403), the addition of two `uint128` numbers may result a number larger than `uint128(-1)`, thus would cause the updated `to.stakingPower` overflow.

Also, we notice the requirement of `(max - min)/10**16 > 0` (line 352) from `vDSGToken::setMintLimitRatio()` is not guarded against possible underflow. It is suggested to replace it with `(max.sub(min))/10**16`

```
> 0.
    function setMintLimitRatio(uint256 min, uint256 max) public onlyOwner {
        require(max < 10**18, "bad max");
        require( (max - min)/10**16 > 0, "bad max - min");

        _MIN_MINT_RATIO_ = min;
        _MAX_MINT_RATIO_ = max;
    }
```

Listing 3.14: `vDSGToken::setMintLimitRatio()`

Recommendation Check if the addition of two `uint128` numbers is larger than `uint128(-1)` to better mitigate possible overflows. Also make use of `SafeMath` in the related calculations to better mitigate possible underflow.

Status The issue has been confirmed by the team. The team clarifies `stakingPower` will not be very large. The possible underflow issue has been fixed by this commit: `53219e5`.

3.9 Safe-Version Replacement With safeTransfer() And safeTransferFrom()

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MysteryBox`
- Category: Coding Practices [10]
- 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 `transfer()` routine and 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.

```
    /**
    * @dev transfer token for a specified address
    * @param _to The address to transfer to.
    * @param _value The amount to be transferred.
    */
    function transfer(address _to, uint _value) public onlyPayloadSize(2 * 32) {
        uint fee = (_value.mul(basisPointsRate)).div(10000);
        if (fee > maximumFee) {
            fee = maximumFee;
```

```
        }
        uint sendAmount = _value.sub(fee);
        balances[msg.sender] = balances[msg.sender].sub(_value);
        balances[_to] = balances[_to].add(sendAmount);
        if (fee > 0) {
            balances[owner] = balances[owner].add(fee);
            Transfer(msg.sender, owner, fee);
        }
        Transfer(msg.sender, _to, sendAmount);
    }
```

Listing 3.15: USDT Token **Contract**

It is important to note the `transfer()` function does not have a return value. However, the `IERC20` interface has defined the following `transfer()` interface with a `bool` return value: `function transfer (address to, uint256 value)external returns (bool)`. 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 `transferFrom()` as well, i.e., `safeTransferFrom()`.

In the following, we show the `emergencyRewardWithdraw()` routine in the `SinglePoolFactory` contract. If `USDT` is given as `token`, the unsafe version of `token.transfer(msg.sender, amount)` (line 161) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value)!

```
155     function emergencyRewardWithdraw(address pool, uint256 _amount) public onlyOwner {
156         IERC20 token = IERC20(SinglePool(pool).rewardToken());
157         uint256 oldAmount = token.balanceOf(address(this));
158         SinglePool(pool).emergencyRewardWithdraw(_amount);
159         uint256 amount = token.balanceOf(address(this)) - oldAmount;

161         require(token.transfer(msg.sender, amount));
162     }
```

Listing 3.16: `SinglePoolFactory::emergencyRewardWithdraw()`

Note that other routine `MysteryBox::buy()` shares the same issue.

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

Status The issue has been fixed by this commit: `2e84184`.

3.10 Possible Sandwich/MEV Attacks For Reduced Returns In Treasury

- ID: PVE-010
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Treasury`
- Category: Time and State [13]
- CWE subcategory: CWE-682 [6]

Description

The `Treasury` contract has a helper routine, i.e., `anySwap()`, that is designed to swap `_amountIn` amount of the `_tokenIn` token to the `_tokenOut` token. It has a rather straightforward logic in calling the `_swap()` to transfer the funds and performing the swap by calling `(ISwapPair(pair).swap)` to actually perform the intended token swap.

```
    function _swap(
        address _tokenIn,
        address _tokenOut,
        uint256 _amountIn,
        address _to
    ) internal returns (uint256 amountOut) {
        address pair = SwapLibrary.pairFor(factory, _tokenIn, _tokenOut);
        (uint256 reserve0, uint256 reserve1, ) = ISwapPair(pair).getReserves();

        (uint256 reserveInput, uint256 reserveOutput) =
            _tokenIn == ISwapPair(pair).token0() ? (reserve0, reserve1) : (reserve1,
                reserve0);
        amountOut = SwapLibrary.getAmountOut(_amountIn, reserveInput, reserveOutput);
        IERC20(_tokenIn).safeTransfer(pair, _amountIn);

        _tokenIn == ISwapPair(pair).token0()
            ? ISwapPair(pair).swap(0, amountOut, _to, new bytes(0))
            : ISwapPair(pair).swap(amountOut, 0, _to, new bytes(0));

        emit Swap(_tokenIn, _tokenOut, _amountIn, amountOut);
    }
```

Listing 3.17: `Treasury::_swap()`

To elaborate, we show above the `_swap()` routine. We notice the actual swap operation via this routine essentially do not specify any restriction on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller return for this round of operation.

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. Note the same issue also exists on the other routines in the `Treasury` contract.

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 team. And the team clarifies that since they will optimize the contract in the future.

3.11 Possible Price manipulation For Oracle::getLpTokenValue()

- ID: PVE-011
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Oracle`
- Category: Time and State [9]
- CWE subcategory: CWE-362 [4]

Description

The `DSG` protocol contains a contract `Oracle` that defines a main function, i.e., `getLpTokenValue()`. This function is used to obtain the price of `LP Token` on the market. During the analysis of the `Oracle::getLpTokenValue()`, we notice the price of `LP Token` is possible to be manipulated. In the following, we show the code snippet of the `getLpTokenValue()` function.

```
    function getLpTokenValue(address _lpToken, uint256 _amount) public view returns (
        uint256 value) {
        uint256 totalSupply = IERC20(_lpToken).totalSupply();
        address token0 = ISwapPair(_lpToken).token0();
        address token1 = ISwapPair(_lpToken).token1();
        uint256 token0Decimal = IERC20p(token0).decimals();
        uint256 token1Decimal = IERC20p(token1).decimals();
        (uint256 reserve0, uint256 reserve1) = SwapLibrary.getReserves(factory, token0,
            token1);

        uint256 token0Value = (getAveragePrice(token0)).mul(reserve0).div(10**
            token0Decimal);
        uint256 token1Value = (getAveragePrice(token1)).mul(reserve1).div(10**
            token1Decimal);
        value = (token0Value.add(token1Value)).mul(_amount).div(totalSupply);
```

```
175     }
```

Listing 3.18: Oracle :: getLpTokenValue()

Specifically, if we examine the implementation of the `getLpTokenValue()`, the final price of the `LP Token` is derived from `(token0Value.add(token1Value)).mul(_amount).div(totalSupply)` (line 174), where each of the `tokenValue` is calculated by `(getAveragePrice(token0)).mul(reserve0)`. Although the price of `token0` or `token1` is the average price from history prices and cannot be manipulated, `reserve0` or `reserve1` is the token amount in the pool thus can be manipulated by flash loans, which cause the final values of the `LP Token` not trustworthy.

Recommendation Revise current execution logic of `getLpTokenValue()` to defensively detect any manipulation attempts in the `LP Token` price.

Status This issue has been confirmed. And the team clarifies that this function will not be used.

3.12 Potential Fund Lockup For Contract Users

- ID: PVE-012
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Business Logic [11]
- CWE subcategory: CWE-841 [7]

Description

As mentioned in Section 3.2, the `DepositPool::deposit()` routine records the depositor's balance when funds are transferred in and the `DepositPool::withdraw()` routine allows the user to redeem the asset by checking the user's balance. To elaborate, we show below the related routines.

```
    function deposit(uint256 _pid, uint256 _amount) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        updatePool(_pid);
        if (user.amount > 0) {
            uint256 pendingAmount = user.amount.mul(pool.accRewardPerShare).div(1e12).
                sub(user.rewardDebt);
            if (pendingAmount > 0) {
                safeRewardTokenTransfer(msg.sender, pendingAmount);
                user.accRewardAmount = user.accRewardAmount.add(pendingAmount);
                pool.allocRewardAmount = pool.allocRewardAmount.sub(pendingAmount);
            }
        }
        if (_amount > 0) {
```

```
213             ERC20(pool.token).safeTransferFrom(msg.sender, address(this), _amount);
214             user.amount = user.amount.add(_amount);
215             pool.totalAmount = pool.totalAmount.add(_amount);
216         }
217         user.rewardDebt = user.amount.mul(pool.accRewardPerShare).div(1e12);
218         emit Deposit(msg.sender, _pid, _amount);
219     }turn true;
220     }
```

Listing 3.19: `DepositPool::deposit()`

```
242     function withdraw(uint256 _pid, uint256 _amount) public {
243         PoolInfo storage pool = poolInfo[_pid];
244         UserInfo storage user = userInfo[_pid][tx.origin];
245         require(user.amount >= _amount, "DepositPool: withdraw: not good");
246         updatePool(_pid);
247         uint256 pendingAmount = user.amount.mul(pool.accRewardPerShare).div(1e12).sub(
                user.rewardDebt);
248         if (pendingAmount > 0) {
249             safeRewardTokenTransfer(tx.origin, pendingAmount);
250             user.accRewardAmount = user.accRewardAmount.add(pendingAmount);
251             pool.allocRewardAmount = pool.allocRewardAmount.sub(pendingAmount);
252         }
253         if (_amount > 0) {
254             user.amount = user.amount.sub(_amount);
255             pool.totalAmount = pool.totalAmount.sub(_amount);
256             ERC20(pool.token).safeTransfer(tx.origin, _amount);
257         }
258         user.rewardDebt = user.amount.mul(pool.accRewardPerShare).div(1e12);
259         emit Withdraw(tx.origin, _pid, _amount);
260     }
```

Listing 3.20: `DepositPool::withdraw()`

We notice the `deposit()` routine is using `msg.sender` to record the depositor balance (line 202) while the `withdraw()` routine is using `tx.origin` to query the depositor's balance (line 244). These two descriptors, `msg.sender` and `tx.origin` will give the same result if an `EOA` is calling the `deposit()` routine directly. However, the values of `msg.sender` and `tx.origin` would be different when the calling is from a contract. Thus, the funds deposited by a contract can not be withdrawn.

Note that other routines `Erc20EarnNftPool::stake()`/`Erc20EarnNftPool::harvest()`, `MysteryBox::buy()`/`MysteryBox::openbox()`, and `TradingPool::swap()`/`TradingPool::withdraw()` share the same issue.

Recommendation Keep consistent of user balance descriptors to avoid failing of withdrawn from contract user.

Status This issue has been confirmed. The team clarifies that for `MysteryBox::buy()`/`MysteryBox::openbox()`, the `box` can be purchased by a contract, but `openBox()` is not allowed. And the user could transfer the `box` to an `EOA` to open it. Other issues have been fixed by the following commits: `50f61bf`, `658020e`, and `04e16d3`.

3.13 Improved Logic of LiquidityPoolOther::emergencyWithdraw()

- ID: PVE-013
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `LiquidityPoolOther`
- Category: Business Logic [11]
- CWE subcategory: CWE-841 [7]

Description

The `LiquidityPoolOther` contract provides an `emergencyWithdraw()` routine that allows staking users to smoothly exit. It is used only in an emergency situation and the calling user has no care of rewards. In the following, we show the `emergencyWithdraw()` routine.

```
    function emergencyWithdraw(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];
        uint256 amount = user.amount;
        user.amount = 0;
        user.rewardDebt = 0;
        IERC20(pool.lpToken).safeTransfer(msg.sender, amount);
        pool.totalAmount = pool.totalAmount.sub(amount);
        emit EmergencyWithdraw(msg.sender, _pid, amount);
    }
```

Listing 3.21: `LiquidityPoolOther.sol::emergencyWithdraw()`

We notice the `emergencyWithdraw()` routine only clears the amount of `user.amount` and `user.rewardDebt`, leaving the `user.additionalAmount` as is!

Recommendation Add `user.additionalAmount = 0` in the `emergencyWithdraw()` routine in the `LiquidityPoolOther.sol` contract.

Status The issue has been fixed by this commit: `f2b6d22`.

4 Conclusion

In this audit, we have analyzed the DSG protocol design and implementation. DSG is a DeFi project built on Binance Smart Chain (BSC). It provides a trading platform and NFT exchange market. During the audit, we notice that the current code base is well organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-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-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.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: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

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

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

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

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

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

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

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

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