SMART CONTRACT AUDIT REPORT

for

Gym Network V2

Prepared By: Xiaomi Huang

PeckShield

October 1, 2022

Document Properties

Client	Gym Network
Title	Smart Contract Audit Report
Target	Gym Network V2
Version	1.0
Author	Xuxian Jiang
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 1, 2022	Xuxian Jiang	Final Release
1.0-rc	August 10, 2022	Shulin Bie	Release Candidate

Contact

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

Name	Xiaomi Huang
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 Gym Network V2, 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 Gym Network V2

Gym Network V2 is a decentralized application that serves as the entry to what the crypto industry has to offer: from DeFi to the metaverse and everything in between. It is based on the BSC (now BNBChain) and it connects to the best interest rates to be found in this blockchain. Its native token gives users the possibility to participate in the governance of the system as well as special access to new features. By connecting the advantages of decentralized systems to the growth potential of affiliate marketing tools, it is an innovative application to bring crypto to the masses.

Table 1.1: Basic Information of The Gym Network

Item	Description
Target	Gym Network V2
Website	https://gymnetwork.io/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 1, 2022

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

- https://gitlab.com/gymnet/mainnet.git (4f752c9)

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

- https://gitlab.com/gymnet/mainnet.git (69d0a60)

1.2 About PeckShield

PeckShield Inc. [12] 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 [11]:

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

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

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

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

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [10], 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 `Gym Network V2` 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	1	■
Low	5	■■■■■
Informational	1	■
Undetermined	1	■
Total	9	

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

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 1 medium-severity vulnerability, 5 low-severity vulnerabilities, 1 informational recommendation, and 1 undetermined issue.

Table 2.1: Key Gym Network V2 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Timely Reward Dissemination Upon Rate Change	Business Logic	Confirmed
PVE-002	High	Improper Logic of GymVaultsBank::_deposit()	Business Logic	Fixed
PVE-003	Low	Improper Logic of GymNetwork::_transferTokens()	Business Logic	Fixed
PVE-004	Low	Timely _moveDelegates() in GymNetwork::burn()	Business Logic	Fixed
PVE-005	Informational	Suggested Event Generation For Key Operations	Coding Practices	Fixed
PVE-006	Undetermined	Incompatibility With Deflationary/Rebasing Tokens	Business Logic	Confirmed
PVE-007	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-008	Low	Revisited Reentrancy Protection in Current Implementation	Time and State	Fixed
PVE-009	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Timely Reward Dissemination Upon Rate Change

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

By design, the `GymFarming` contract implements an incentive mechanism that rewards the staking of supported assets with the `rewardToken` token. The rewards are carried out by designating a number of staking pools. The staking users are rewarded in proportional to their staking assets in the pool.

The reward rate (per block) of the `rewardToken` token can be adjusted via the `GymFarming::setRewardConfiguration()` routine. When analyzing its logic, we notice the lack of timely invoking `massUpdatePools()` to update each pool reward status before the new reward-related configuration becomes effective. If the call to `massUpdatePools()` is not immediately invoked before updating the reward rate, certain situations may be crafted to create an unfair reward distribution.

```
    function setRewardConfiguration(uint256 _rewardPerBlock, uint256
        _rewardUpdateBlocksInterval)
        external
        onlyOwner
    {
        _setRewardConfiguration(_rewardPerBlock, _rewardUpdateBlocksInterval);
    }
```

Listing 3.1: `GymFarming::setRewardConfiguration()`

```
    function _setRewardConfiguration(uint256 rewardPerBlock, uint256
        updateBlocksInterval)
        internal
    {
        uint256 oldRewardValue = rewardsConfiguration.rewardPerBlock;
```

```

        rewardsConfiguration.rewardPerBlock = rewardPerBlock;
        rewardsConfiguration.lastUpdateBlockNum = block.number;
        rewardsConfiguration.updateBlocksInterval = updateBlocksInterval;

        emit RewardPerBlockUpdated(oldRewardValue, rewardPerBlock);
    }
```

Listing 3.2: `RewardRateConfigurable::_setRewardConfiguration()`

Note other routines, i.e., `GymFarming::add()/setRewardConfiguration()/updatePool()`, `RewardRateConfigurable::_updateRewardPerBlock()`, `GymSinglePool::setPoolInfo()/updatePool()` and `GymVaultsBank::setRewardConfiguration()/addPool()/updatePool()`, are also influenced by this issue.

Recommendation Timely invoke `massUpdatePools()` before the new reward-related configuration becomes effective.

Status The issue has confirmed by the team.

3.2 Improper Logic Of GymVaultsBank::_deposit()

- ID: PVE-002
- Severity: High
- Likelihood: High
- Impact: High
- Target: `GymVaultsBank`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

By design, the `GymVaultsBank` contract is one of the main entries for interaction with users. In particular, it implements an incentive mechanism that rewards the staking of supported assets with the `rewardToken` token. One specific entry routine, i.e., `deposit()`, is designed to deposit the supported assets. While examining its logic, we observe there is a vulnerability that can be exploited by the malicious actor to claim the reward repeatedly.

To elaborate, we show below the related code snippet of the `GymVaultsBank` contract. At the beginning of the internal `_deposit()` routine (which is called inside the `deposit()` routine), the `_claim()` routine is invoked (line 433) to calculate and transfer the pending rewards to `msg.sender`. Additionally, we notice the `_claim()` routine is not in charge of `user.rewardDebt` update. That is to say, the `user.rewardDebt` should be updated outside the `_claim()` routine. After further analysis, we notice the `user.rewardDebt` is updated (line 456) only when the deposit amount is larger than 0 (line 436). With that, a malicious actor can claim the reward repeatedly by making the deposit amount 0. Given this, we suggest to update the `user.rewardDebt` regardless the actual deposit amount.

```
    function deposit(
        uint256 _pid,
        uint256 _wantAmt,
        uint256 _referrerId
    ) external payable nonReentrant onlyEmailVerified(msg.sender) {
        ...

        _deposit(_pid, _wantAmt);
        _updateLevelPoolQualification(msg.sender);
    }

    function _claim(uint256 _pid, address _user) private {
        PoolInfo memory pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][_user];
        uint256 pending = (user.shares * pool.accRewardPerShare) / (1e18) - (user.
            rewardDebt);
        if (pending > 0) {
            uint256 _distributedRewards = _distributeRewards(pending, rewardToken, _user
                );
            user.totalClaims += (pending - _distributedRewards);
            _safeRewardTransfer(rewardToken, _user, (pending - _distributedRewards));
            emit RewardPaid(rewardToken, _user, pending);
        }
    }

    function _deposit(uint256 _pid, uint256 _wantAmt) private {
        updatePool(_pid);
        PoolInfo memory pool = poolInfo[_pid];
        UserInfo storage user = userInfo[_pid][msg.sender];

        if (user.shares > 0) {
            _claim(_pid, msg.sender);
        }

        if (_wantAmt > 0) {
            if (msg.value == 0  address(pool.want) != wbnbAddress) {
                // If ‘want‘ not WBNB
                pool.want.safeTransferFrom(address(msg.sender), address(this), _wantAmt)
                    ;
            }

            if (address(pool.want) == busdAddress) {
                user.dollarValue += (_wantAmt / 1e18);
            } else if (address(pool.want) == wbnbAddress) {
                user.dollarValue += ((_wantAmt * IGYMNETWORK(gymNetworkAddress).
                    getBNBPrice()) /
                    1e18);
            }

            pool.want.safeIncreaseAllowance(pool.strategy, _wantAmt);
            uint256 sharesAdded = IStrategy(poolInfo[_pid].strategy).deposit(msg.sender,
                 _wantAmt);
```

```
            user.shares += sharesAdded;

            _updateInvestment(msg.sender);
            userInvestment[_pid][msg.sender] += _wantAmt;

            user.rewardDebt = (user.shares * (pool.accRewardPerShare)) / (1e18);

            emit Deposit(msg.sender, _pid, _wantAmt);
        }
    }
```

Listing 3.3: `GymVaultsBank::deposit()`

Recommendation Correct the implementation of the `_deposit()` routine as above-mentioned.

Status The issue has been addressed by the following commit: `fb32bb2`.

3.3 Improper Logic of GymNetwork::_transferTokens()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `GymNetwork`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The `GymNetwork` contract implements a so-called deflationary token that charges a certain fee for every token transfer (via `transfer()` or `transferFrom()`). Meanwhile, the privileged `owner` can configure the special addresses that exclude from charging fee. In particular, the internal `_transferTokens()` routine is called inside the `transfer()` and `transferFrom()` routines to transfer the `GYNNET` token with each other. While examining its logic, we notice there is an improper implementation that needs to be improved.

To elaborate, we show below the related code snippet of the `GymNetwork` contract. By design, if the token sender or recipient is the special `taxCollector`, there is no need to charge fee. The following `if` statement (i.e., `if ((!isDex[dst] && !isDex[src])|| (_dexTaxExcempt[dst] || _dexTaxExcempt[src])|| src == taxCollector || src == taxCollector)` (lines 384 - 389)) is designed to meet the requirement. However, it comes to our attention that the check of `src == taxCollector` is duplicate (lines 387/388) in the `if` statement. We believe one of them should be the check on `dst == taxCollector`.

```
375     function _transferTokens(
376         address src,
377         address dst,
378         uint96 amount
```

```
    ) internal {
        ...

        uint96 maxPerHolder = (totalSupply * MAX_PER_HOLDER_PERCENT) / 100;

        if (
            (!isDex[dst] && !isDex[src])
            (_dexTaxExcempt[dst]   _dexTaxExcempt[src])
            src == taxCollector
            src == taxCollector
        ) {
            if (!_isLimitExcempt[dst]) {
                require(
                    add96(
                        balances[dst],
                        amount,
                        "GymNet::_transferTokens: exceds max per holder amount"
                    ) <= maxPerHolder,
                    "GymNet::_transferTokens: final balance exceeds balance limit"
                );
            }
            balances[src] = sub96(
                balances[src],
                amount,
                "GymNet::_transferTokens: transfer amount exceeds balance"
            );
            balances[dst] = add96(
                balances[dst],
                amount,
                "GymNet::_transferTokens: transfer amount overflows"
            );
            emit Transfer(src, dst, amount);

            _moveDelegates(delegates[src], delegates[dst], amount);
        } else {
            ...
        }
    }
```

Listing 3.4: `GymNetwork::_transferTokens()`

Recommendation Correct the implementation of the above-mentioned `_transferTokens()` routine.

Status The issue has been addressed by the following commit: `440b103`.

3.4 Timely _moveDelegates() in GymNetwork::burn()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: GymNetwork
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The GYMNET token can be used for governance in allowing for users to cast and record the votes. Moreover, the GymNetwork contract allows for dynamic delegation of a voter to another, though the delegation is not transitive. When a submitted proposal is being tallied, the number of votes are counted via getPriorVotes().

When analyzing the burn() routine of the GYMNET token, we notice the need of timely invoking _moveDelegates() to update voting power delegation. In the current implementation, even if the user's GYMNET token is burned, his voting power delegation still takes effect, which directly undermines the design.

```
    function burn(uint256 rawAmount) public {
        uint96 amount = safe96(rawAmount, "GymNet::approve: amount exceeds 96 bits");
        _burn(msg.sender, amount);
    }

    function burnFrom(address account, uint256 rawAmount) public {
        uint96 amount = safe96(rawAmount, "GymNet::approve: amount exceeds 96 bits");
        uint96 currentAllowance = allowances[account][msg.sender];
        require(currentAllowance >= amount, "GymToken: burn amount exceeds allowance");
        allowances[account][msg.sender] = currentAllowance - amount;
        _burn(account, amount);
    }

    function mintFor(address account, uint96 amount) public onlyOwner {
        require(
            minted + amount <= MAX_SUPPLY,
            "GymNet::_adminFunctions: Mint more tokens than allowed"
        );

        totalSupply += amount;
        minted += amount;

        balances[account] += uint96(amount);
        emit Transfer(address(0), account, amount);
    }
```

Listing 3.5: GymNetwork

Note other routines, i.e., burnFrom()/mintFor(), can be similarly improved.

Recommendation Timely invoke _moveDelegates() to update voting power delegation in above-mentioned routines.

Status The issue has been addressed by the following commits: 51645cc and e053ba3.

3.5 Suggested Event Generation For Key Operations

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contracts
- Category: Coding Practices [7]
- CWE subcategory: CWE-563 [3]

Description

In Ethereum, the event is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an event is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

While examining the events that reflect the protocol dynamics, we notice there are several key operations that lack meaningful events to reflect their changes. In the following, we show several representative routines.

```
    function setRewardToken(address _rewardToken) external onlyOwner {
        rewardToken = _rewardToken;
        rewardTokenToWBNB = [_rewardToken, wbnbAddress];
    }

    function setMLMAddress(address _address) external onlyOwner validAddress(_address) {
        relationship = _address;
    }

    function setLevelPoolAddress(address _levelPoolAddress)
        external
        onlyOwner
        validAddress(_levelPoolAddress)
    {
        levelPool = _levelPoolAddress;
    }

    function setTreasuryAddress(address _newTreasury)
        external
```

```
        onlyOwner
        validAddress(_newTreasury)
    {
        treasury = _newTreasury;
    }
```

Listing 3.6: GymFarming

With that, we suggest to emit meaningful events for these key operations. Also, the key event information is better indexed. Note each emitted event is represented as a topic that usually consists of the signature (from a keccak256 hash) of the event name and the types (uint256, string, etc.) of its parameters. Each indexed type will be treated like an additional topic. If an argument is not indexed, it will be attached as data (instead of a separate topic). Considering that the key information is typically queried, it is better treated as a topic, hence the need of being indexed.

Recommendation Properly emit the above-mentioned events with accurate information to timely reflect state changes. This is very helpful for external analytics and reporting tools.

Status The issue has been addressed by the following commit: fb32bb2.

3.6 Incompatibility With Deflationary/Rebasing Tokens

- ID: PVE-006
- Severity: Undetermined
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contracts
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

In the Gym Network V2 protocol, the GymSinglePool contract is one of the main entries for interaction with users. In particular, one entry routine, i.e., deposit(), accepts the deposits of the supported tokenAddress token. Naturally, the contract implements a number of low-level helper routines to transfer assets into or out of the protocol. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract.

```
    function deposit(
        uint256 _depositAmount,
        uint8 _periodId,
        bool isUnlocked
    ) external hasInvestment(msg.sender) {
        require(isPoolActive, "Contract is not running yet");
        //TO-DO Add Vault check here
```

```
        if (isUnlocked) {
            _periodId = 0;
        }
        _deposit(_depositAmount, _periodId, isUnlocked);
        _updateLevelPoolQualification(msg.sender);
    }

    function _deposit(
        uint256 _depositAmount,
        uint8 _periodId,
        bool _isUnlocked
    ) private {
        UserInfo storage user = userInfo[msg.sender];
        IERC20Upgradeable token = IERC20Upgradeable(tokenAddress);
        PoolInfo storage pool = poolInfo;
        updatePool();

        uint256 lockTimesamp = DateTime.addMonths(block.timestamp, months[_periodId]);
        uint256 burnTokensAmount = 0;

        if (!_isUnlocked) {
            burnTokensAmount = (_depositAmount * 4) / 100;
            totalBurntInSinglePool += burnTokensAmount;
            IERC20Burnable(tokenAddress).burnFrom(msg.sender, burnTokensAmount);
        }

        uint256 amountToDeposit = _depositAmount - burnTokensAmount;

        token.safeTransferFrom(msg.sender, address(this), amountToDeposit);
        uint256 UsdValueOfGym = ((amountToDeposit * IGYMNETWORK(tokenAddress).
            getGYMNETPrice()) /
            1e18) / 1e18;
        uint256 _ggymnetAmt = (amountToDeposit * ggymnetAlloc[_periodId]) / 1e18;

        if (_isUnlocked) {
            _ggymnetAmt = 0;
            totalGymnetUnlocked += amountToDeposit;
            lockTimesamp = DateTime.addSeconds(block.timestamp, months[_periodId]);
        }
        user.totalDepositTokens += amountToDeposit;
        user.totalDepositDollarValue += UsdValueOfGym;
        totalGymnetLocked += amountToDeposit;
        totalGGymnetInPoolLocked += _ggymnetAmt;

        uint256 rewardDebt = (_ggymnetAmt * (pool.accRewardPerShare)) / (1e18);
        UserDeposits memory depositDetails = UserDeposits({
            depositTokens: amountToDeposit,
            depositDollarValue: UsdValueOfGym,
            stakePeriod: _isUnlocked ? 0 : months[_periodId],
            depositTimestamp: block.timestamp,
            withdrawalTimestamp: lockTimesamp,
            rewardsGained: 0,
```

```
            is_finished: false,
            rewardsClaimt: 0,
            rewardDebt: rewardDebt,
            ggymnetAmt: _ggymnetAmt,
            is_unlocked: _isUnlocked
        });
        user.totalGGYMNET += _ggymnetAmt;
        user_deposits[msg.sender].push(depositDetails);
        user.depositId = user_deposits[msg.sender].length;

        refreshMyLevel(msg.sender);
        emit Deposit(msg.sender, _depositAmount, _periodId);
    }
```

Listing 3.7: `GymSinglePool::deposit()`

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every `transfer()` or `transferFrom()`. (Another type is rebasing tokens such as `YAM`.) As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as `deposit()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of expecting the amount parameter in `transfer()` or `transferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the contract before and after the `transfer()` or `transferFrom()` 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.

Note that other routines related to token transfer, e.g., `GymFarming::deposit()` and `GymVaultsBank::deposit()`, share the similar issue.

Recommendation If current codebase needs to support deflationary tokens, it is necessary to check the balance before and after the `transfer()/transferFrom()` call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is the widely-adopted `USDT`.

Status The issue has been confirmed by the team. There is no need to support deflationary/rebasing tokens.

3.7 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [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 stablecoin, i.e., USDT, as our example. We show the related code snippet below. Specifically, the transfer() routine does not have a return value defined and implemented. However, the IERC20 interface has defined the transfer() interface with a bool return value. As a result, the call to transfer() may expect a return value. With the lack of return value of USDT's transfer(), the call may be unfortunately reverted.

```
    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 3.8: USDT Token Contract

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

In the following, we show below the GymSinglePool::safeRewardTransfer() routine. If the USDT token is supported as _rewardToken, the unsafe version of require(IERC20Upgradeable(_rewardToken) .transfer(_to, _bal), "GymSinglePool:: Transfer failed") (lines 625 - 628) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation. We may intend to replace `require(IERC20Upgradeable(_rewardToken).transfer(_to, _bal), "GymSinglePool:: Transfer failed")` (lines 625 - 628) with `safeTransfer()`.

```
618     function safeRewardTransfer(
619         address _rewardToken,
620         address _to,
621         uint256 _amount
622     ) internal {
623         uint256 _bal = IERC20Upgradeable(_rewardToken).balanceOf(address(this));
624         if (_amount > _bal) {
625             require(
626                 IERC20Upgradeable(_rewardToken).transfer(_to, _bal),
627                 "GymSinglePool:: Transfer failed"
628             );
629         } else {
630             require(
631                 IERC20Upgradeable(_rewardToken).transfer(_to, _amount),
632                 "GymSinglePool:: Transfer failed"
633             );
634         }
635     }
```

Listing 3.9: `GymSinglePool::safeRewardTransfer()`

Note that all the routines using `approve()/transfer()` can be similarly improved.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `transfer()/approve()`. And there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

Status The issue has been addressed by the following commit: `fb32bb2`.

3.8 Revisited Reentrancy Protection in Current Implementation

- ID: PVE-008
- Severity: Medium
- Likelihood: Low
- Impact:High
- Target: `Multiple Contracts`
- Category: Time and State [9]
- CWE subcategory: CWE-682 [4]

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` [14] exploit, and the recent `Uniswap/Lendf.Me` hack [13].

In the following, we use the `GymAccountant::claimPendingRewards()` routine as an example. To elaborate, we show below the related code snippet of the contract. Inside the `_claimPendingRewards()` routine, we notice `require(IERC20Upgradeable(tokenAddress).transfer(_userAddress, _pendingRewards), "GymAccountant:: Transfer failed")` is called to transfer the pending rewards to the recipient. If the `tokenAddress` faithfully implements the ERC777-like standard, then the `claimPendingRewards()` routine is vulnerable to reentrancy and this risk needs to be properly mitigated.

Specifically, the ERC777 standard normalizes the ways to interact with a token contract while remaining backward compatible with ERC20. Among various features, it supports send/receive hooks to offer token holders more control over their tokens. Specifically, when `transfer()` or `transferFrom()` actions happen, the owner can be notified to make a judgment call so that she can control (or even reject) which token they send or receive by correspondingly registering `tokensToSend()` and `tokensReceived()` hooks. Consequently, any `transfer()` or `transferFrom()` of ERC777-based tokens might introduce the chance for reentrancy or hook execution for unintended purposes (e.g., mining `GasTokens`).

In our case, the above hook can be planted in `require(IERC20Upgradeable(tokenAddress).transfer(_userAddress, _pendingRewards), "GymAccountant:: Transfer failed")` before the actual transfer of the underlying assets occurs. By doing so, we can effectively keep `borrowedAmountByType[_type][_userAddress]` intact (used for the calculation of pending rewards at line 69). With a lower `borrowedAmountByType[_type][_userAddress]`, the re-entered `claimPendingRewards()` is able to obtain more rewards. It can be repeated to exploit this vulnerability for gains.

```
63     function claimPendingRewards(uint32 _type) external {
64         require(
65             mlmAddress != address(0),
66             "GymAccountant:: GymMLM contract address is zero address"
67         );
68 
69         uint256 _pendingRewards = IGymMLM(mlmAddress).getPendingRewards(msg.sender,
               _type);
70 
71         require(_pendingRewards > 0, "GymAccountant:: Zero pending rewards");
72         _claimPendingRewards(msg.sender, _pendingRewards, _type);
73     }
74 
75     function _claimPendingRewards(
76         address _userAddress,
77         uint256 _pendingRewards,
78         uint256 _type
79     ) private {
```

```
        require(tokenAddress != address(0), "GymAccountant:: Token address is zero");
        require(
            IERC20Upgradeable(tokenAddress).transfer(_userAddress, _pendingRewards),
            "GymAccountant:: Transfer failed"
        );
        _updateBorrowedAmount(_userAddress, _pendingRewards, _type, true);
    }

    function _updateBorrowedAmount(
        address _userAddress,
        uint256 _amount,
        uint256 _type,
        bool _increase
    ) private {
        if (_increase) {
            borrowedAmountByType[_type][_userAddress] += _amount;
        } else {
            borrowedAmountByType[_type][_userAddress] -= _amount;
        }

        uint256 returnAmount = borrowedAmountByType[_type][_userAddress];
        emit BorrowedAmountUpdated(_userAddress, _type, _amount, returnAmount);
    }
```

Listing 3.10: `GymAccountant::claimPendingRewards()`

Moreover, we observe most routines in the `Gym Network V2` protocol haven't considered reentrancy protection.

Recommendation Add necessary reentrancy guards to prevent unwanted reentrancy risks.

Status The issue has been addressed in the latest version.

3.9 Trust Issue of Admin Keys

- ID: PVE-009
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [2]

Description

The `Gym Network V2` protocol has a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., pool addition, reward adjustment, and parameter setting). In the following, we show the representative functions potentially affected by the privilege of the account.

```
    function updateTaxCollector(address _taxCollector) public onlyOwner {
        taxCollector = _taxCollector;
    }

    function manageBlacklist(address[] memory users, bool[] memory _toBlackList) public
        onlyOwner {
        require(users.length == _toBlackList.length, "GymNet::_adminFunctions: Array
            mismatch");

        for (uint256 i; i < users.length; i++) {
            _isBlackListed[users[i]] = _toBlackList[i];
        }
    }

    function manageSellLimitExcempt(address[] memory users, bool[] memory _toLimit)
        public
        onlyOwner
    {
        require(users.length == _toLimit.length, "GymNet::_adminFunctions: Array
            mismatch");

        for (uint256 i; i < users.length; i++) {
            _isSellLimited[users[i]] = _toLimit[i];
        }
    }

    function mintFor(address account, uint96 amount) public onlyOwner {
        require(
            minted + amount <= MAX_SUPPLY,
            "GymNet::_adminFunctions: Mint more tokens than allowed"
        );

        totalSupply += amount;
        minted += amount;

        balances[account] += uint96(amount);
        emit Transfer(address(0), account, amount);
    }
```

Listing 3.11: Example Privileged Operations in `GymNetwork`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged account is not governed by a DAO-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

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.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Gym Network V2`, which is a decentralized application that serves as the entry to what the crypto industry has to offer: from `DeFi` to the `metaverse` and everything in between. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

Meanwhile, 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-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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

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

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

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

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

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

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

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

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

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

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