SMART CONTRACT AUDIT REPORT

for

GymStreet

Prepared By: Xiaomi Huang

PeckShield

September 29, 2022

Document Properties

Client	Gym Network
Title	Smart Contract Audit Report
Target	GymStreet
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	September 29, 2022	Xuxian Jiang	Final Release
1.0-rc	September 2, 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 GymStreet 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 GymStreet

GymStreet is the first purpose-built DeFi and CeFi metaverse: a virtual realm of experienced finance with multiple types of NFTs and 2.5D (later 3D) graphics. GymStreet is GameFi in its true sense instead of simply adding DeFi elements into a game. GymStreet turns DeFi and CeFi into a game with high-quality graphics and animation. The metaverse will even have a conversational AI, ready to answer users' questions.

Table 1.1: Basic Information of GymStreet

Item	Description
Target	GymStreet
Website	http://gymstreet.io/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 29, 2022

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

- https://gitlab.com/gymstreet/smart-contracts.git (15b16f2)

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

- https://gitlab.com/gymstreet/smart-contracts.git (ee98cc0)

1.2 About PeckShield

PeckShield Inc. [10] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [9]:

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

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

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) [8], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

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 `GymStreet` 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	2	
Informational	1	
Total	6	

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, 2 medium-severity vulnerabilities, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key GymStreet Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Improper Logic of ERC721Base::_addTokenTo()/_removeTokenFrom()	Business Logic	Fixed
PVE-002	Informational	Suggested Event Generation for Key Operations	Coding Practices	Fixed
PVE-003	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-005	Low	Timely Reward Dissemination upon Rate Change	Business Logic	Fixed
PVE-006	Medium	Improper Logic of Mining::getDateTimeConcat()	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 Improper Logic of ERC721Base::_addTokenTo()/_removeTokenFrom()

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: `ERC721Base`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `ERC721Base` contract implements the standard `ERC721` interfaces. Additionally, it implements the enumerability of all token IDs owned by the user. In particular, the `mapping(address => uint256[])internal _userTokens` is designed to record all the token IDs held by the user and the `mapping(uint256 => uint256)internal _indexOfToken` records the index of the token inside the user's token array. Meanwhile, the `_addTokenTo()` and `_removeTokenFrom()` routines are designed to manage the token IDs held by the user. While examining the related logic, we observe the current implementation should be improved.

To elaborate, we show below the related code snippet of the `ERC721Base` contract. Inside the `_addTokenTo()` routine, the statement of `_userTokens[_to].push(_tokenId)` (line 407) is executed to push the `_tokenId` to the `_to`'s token array. Subsequently, the statement of `_indexOfToken[_tokenId] = _userTokens[_to].length` (line 408) is executed to record the index of the `_tokenId` inside the user's token array. However, it ignores the fact that the index of the array starts from 0.

```
405     function _addTokenTo(address _to, uint256 _tokenId) internal {
406         _tokenOwner[_tokenId] = _to;
407         _userTokens[_to].push(_tokenId);
408         _indexOfToken[_tokenId] = _userTokens[_to].length;
409         _tokensCount = _tokensCount.add(1);
410         _userPurchaseDate[_to] = block.timestamp;
```

```
    }
```

Listing 3.1: `ERC721Base::_addTokenTo()`

Moreover, by design, the `_removeTokenFrom()` routine is used to remove the given `_tokenId` token from the given `_from` address. In order to meet the requirement, it needs to replace `_tokenId` with the last token ID inside the `_from`'s token array and update the index of the last token ID. Eventually, the array's last element should be released via `pop()`. However, it comes to our attention that the current implementation is far from the design.

```
    function _removeTokenFrom(address _from, uint256 _tokenId) internal {
        uint256 tokenIndex = _indexOfToken[_tokenId];
        uint256 lastTokenIndex = _userTokens[_from].length.sub(1);
        uint256 lastTokenId = _indexOfToken[lastTokenIndex];

        _userTokens[_from][tokenIndex] = lastTokenId;
        _indexOfToken[lastTokenId] = tokenIndex;
        _userTokens[_from].pop();

        _tokenOwner[_tokenId] = address(0);
        _tokensCount = _tokensCount.sub(1);

        if (_userTokens[_from].length == 0) {
            delete _userTokens[_from];
        }
    }
```

Listing 3.2: `ERC721Base::_removeTokenFrom()`

Recommendation Correct the implementation of above-mentioned routines as below:

```
    function _addTokenTo(address _to, uint256 _tokenId) internal {
        _tokenOwner[_tokenId] = _to;
        _userTokens[_to].push(_tokenId);
        _indexOfToken[_tokenId] = _userTokens[_to].length - 1;
        _tokensCount = _tokensCount.add(1);
        _userPurchaseDate[_to] = block.timestamp;
    }
```

Listing 3.3: `ERC721Base::_addTokenTo()`

```
    function _removeTokenFrom(address _from, uint256 _tokenId) internal {
        uint256 tokenIndex = _indexOfToken[_tokenId];
        uint256 lastTokenId = _userTokens[_from][_userTokens[_from].length - 1];

        _userTokens[_from][tokenIndex] = lastTokenId;
        delete _indexOfToken[_tokenId];
        _userTokens[_from].pop();

        _tokenOwner[_tokenId] = address(0);
        _tokensCount = _tokensCount.sub(1);
```

```
        if (_userTokens[_from].length == 0) {
            delete _userTokens[_from];
        }
    }
```

Listing 3.4: `ERC721Base::_removeTokenFrom()`

Status The issue has been addressed by the following commit: `598e5d9`.

3.2 Suggested Event Generation For Key Operations

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [6]
- 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 setStandardParcelAddress(address _contract) external onlyOwner {
        standardParcelAddress = _contract;
    }

    function setBusinessParcelAddress(address _contract) external onlyOwner {
        businessParcelAddress = _contract;
    }

    function setMinerAddress(address _contract) external onlyOwner {
        minerNFTAddress = _contract;
    }

    function setMLMQualificationsAddress(address _address) external onlyOwner {
        mlmQualificationsAddress = _address;
    }
```

Listing 3.5: `NetGymStreet`

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: `0fd686a1`.

3.3 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `NetGymStreet`
- Category: Coding Practices [6]
- 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 3.6: `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.

In the following, we show the `distributeRewards()` routine. If the `USDT` token is supported as `token`, the unsafe version of `require(token.transfer(_referrers[index], rewardToTransfer), "NetGymStreet:: Transfer failed")` (lines 180 - 183) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation. We may intend to replace `require(token.transfer(_referrers[index], rewardToTransfer), "NetGymStreet:: Transfer failed")` (lines 180 - 183) with `safeTransfer()`.

```
163     function distributeRewards(
164         uint256 _wantAmt,
165         address _wantAddr,
166         address _user
167     ) external onlyMunicipality {
168         uint256 index;
169         uint256 rewardToTransfer;
170         IERC20Upgradeable token = IERC20Upgradeable(_wantAddr);
171 
172         address[] memory _referrers = IGymMLM(mlmAddress).getReferrals(_user);
173 
174         while (index < directReferralBonuses.length && index < _referrers.length) {
175             uint256 _level = _getUserLevel(_referrers[index]);
176 
177             if (index <= _level && hasNFT(_user)) {
178                 rewardToTransfer = (_wantAmt * directReferralBonuses[index]) / 10000;
179 
180                 require(
181                     token.transfer(_referrers[index], rewardToTransfer),
182                     "NetGymStreet:: Transfer failed"
183                 );
184                 ...
185             }
186 
187             rewardToTransfer = 0;
188             index++;
189         }
190         ...
191     }
```

Listing 3.7: `NetGymStreet::distributeRewards()`

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

Status The issue has been addressed by the following commit: `20f87c5f`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

The `GymStreet` protocol has a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configuring various system parameters). In the following, we show the representative functions potentially affected by the privilege of the account.

```
    function setTransferActivation(bool _transferActivation) public onlyOwner {
        _setTransferActivation(_transferActivation);
        emit TransferActivationSet(_transferActivation);
    }

    function setMaxSupply(uint256 _maxSupply) external onlyOwner {
        _setMaxSupply(_maxSupply);
        emit MaxSupplySet(_maxSupply);
    }

    function setMunicipalityAddress(address _municipalityAddress) external onlyOwner {
        municipalityAddress = _municipalityAddress;
        emit MunicipalityAddressSet(municipalityAddress);
    }

    function setMinerPublicBuildingAddress(address _minerPublicBuilding) external
        onlyOwner {
        minerPublicBuilding = _minerPublicBuilding;
        emit MinerPublicBuildingSet(minerPublicBuilding);
    }

    /// @notice IParcelInterface functions
    function mint(address _user, uint256 _x, uint256 _y, uint256 _lt) public
        onlyAuthorizedContracts returns (uint256) {
        uint256 parcelId = _getParcelId(_x, _y, _lt);
        require(!_exists(parcelId), "StandardParcelNFT: Parcel already exists as a
            standard parcel");
        _mintFor(parcelId, _user);
        return parcelId;
```

```
    }
```

Listing 3.8: Example Privileged Operations in `StandardParcelNFT`

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.

3.5 Timely Reward Dissemination upon Rate Change

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Mining`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

By design, the `Mining` contract implements an incentive mechanism that rewards the `miners` with the `rewardToken` token. The `miners` are rewarded in proportional to their `hashrate` in the pool.

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

```
    function setRewardPerBlock(uint256 _rewardPerBlock) external onlyOwner {
        rewardPerBlock = _rewardPerBlock;
    }
```

Listing 3.9: `Mining::setRewardPerBlock()`

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

Status The issue has been addressed by the following commit: `5bfd35ed`.

3.6 Improper Logic of Mining::getDateTimeConcat()

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Mining`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

By design, the `Mining` contract is one of the main entries, which implements an incentive mechanism that rewards the `miners` with the `rewardToken` token. In particular, the `mapping(uint256 => uint256) public rewardSharesByDays` is designed to store the pool's `accRewardPerShare` per day. Meanwhile, the `getDateTimeConcat()` routine is used to generate the key of the `rewardSharesByDays` mapping. While examining its logic, we observe there is an improper implementation that needs to be improved.

To elaborate, we show below the related code snippet of the `Mining` contract. Inside the `getDateTimeConcat()` routine, the formula of `uint256 date = (year * 1000)+ (month * 100)+ day` (line 410) is designed to generate the key of the `rewardSharesByDays` mapping to represent one day uniquely. However, after further analysis, we observe different days may generate the same key (e.g., **2022** * 1000 + **11** * 100 + **1** == **2023** * 1000 + **1** * 100 + **1**), which directly undermines the assumption of the design. Given this, we suggest to improve the formula as below: `uint256 date = (year * 10000)+ (month * 100)+ day` (line 410).

```
408     function getDateTimeConcat(uint256 _timestamp) public pure returns (uint256) {
409         (uint256 year,uint256 month,uint256 day) = DateTime.timestampToDate(_timestamp);
410         uint256 date = (year * 1000) + (month * 100) + day;
411         return date;
412     }
```

Listing 3.10: `Mining::getDateTimeConcat()`

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

Status The issue has been addressed by the following commit: `5bfd35e`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `GymStreet`, which is the first purpose-built `DeFi` and `CeFi` metaverse: a virtual realm of experienced finance with multiple types of NFTs and 2.5D (later 3D) graphics. `GymStreet` is `GameFi` in its true sense instead of simply adding `DeFi` elements into a game. `GymStreet` turns `DeFi` and `CeFi` into a game with high-quality graphics and animation. 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-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

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

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

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

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