SMART CONTRACT AUDIT REPORT

for

PLAYPAD-IDO-DQ

Prepared By: Yiqun Chen

PeckShield

November 14, 2021

Document Properties

Client	PlayPad
Title	Smart Contract Audit Report
Target	PlayPad-IDO-DQ
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 14, 2021	Xiaotao Wu	Final Release

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 PlayPad 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 PlayPad-IDO-DQ

In most known Tier systems, only the number of tokens staked by each participant is taken into account to take part in an IGO. In PlayPad day and quantity model allocation system, the day staked is as important as the number of tokens staked. The PlayPad DQ moves away from the uniform standardized model for IGO participation and allows investors to get vestings with less stakes. In PlayPad DQ, investors' stake time also plays an active role in distribution. Staking time will reduce investors' post-IGO sales pressure as it increases the chances of participation.

The basic information of audited contracts is as follows:

Table 1.1: Basic Information of PlayPad-IDO-DQ

Item	Description
Name	PlayPad
Website	https://playpad.app/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 14, 2021

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

- https://github.com/PlayPad0/PlayPad-IDO-DQ/blob/main/playPadIdoMain.sol (dc573ef)

1.2 About PeckShield

PeckShield Inc. [7] 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 [6]:

- 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 accordingly classified into four categories, 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 contract is considered safe regarding the check item. For any discovered issue, we might further

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

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) [5], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business 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.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the `PlayPad-IDO-DQ` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	3	■ ■ ■
Low	1	■
Informational	0	
Total	4	

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

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 3 medium-severity vulnerabilities, and 1 low-severity vulnerability.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improved Logic In MainPlayPadContract::stakeTokens()	Business Logic	Fixed
PVE-002	Medium	Improved Logic In MainPlayPadContract::withdrawPoolRemainder()	Business Logic	
PVE-003	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Improved Logic In MainPlayPadContract::stakeTokens()

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `MainPlayPadContract`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The `PlayPad-IDO-DQ` protocol users can stake their `stakingToken` to the `MainPlayPadContract` contract to earn rewards. While examining the `stakeTokens()` routine of the `MainPlayPadContract` contract, we notice the current implementation logic can be improved.

To elaborate, we show below its code snippet. It comes to our attention that when staking `stakingToken` to `MainPlayPadContract`, the `user.stakeStartDate` state will be updated every time if the total staked amount of a staker is above the limit for vesting, i.e., `limitForPrize` (lines 1074). However, the `user.stakeStartDate` of the staker should only be updated when the total staked amount of this staker is above the vesting limit for the first time.

```
1031     //stake tokens with controls
1032     function stakeTokens(uint256 _amountToStake) external nonReentrant {
1033         updatePool();
1034         uint256 pending = 0;
1035         uint256 randomPoolId =
1036             uint256(
1037                 keccak256(
1038                     abi.encodePacked(
1039                         msg.sender,
1040                         now,
1041                         block.number,
1042                         _amountToStake
1043                     )
1044                 )
1045             );
```

```
        require(!availablePools[randomPoolId], "Pool id already created");
        UserInfo storage user = userInfo[msg.sender];
        UserPoolInfo storage poolInfo = userPoolInfo[randomPoolId];
        if (user.amount > 0) {
            pending = transferPendingReward(user);
        } else if (_amountToStake > 0) {
            participants += 1;
        }

        if (_amountToStake > 0) {
            stakingToken.safeTransferFrom(
                msg.sender,
                address(this),
                _amountToStake
            );
            if(user.userAddress == address(0x0000000000000000000000000000000000000000)){
                allInvestors.push(msg.sender);
            }
            availablePools[randomPoolId] = true;
            poolInfo.blockNumber = block.number;
            poolInfo.amount = _amountToStake;
            poolInfo.owner = msg.sender;
            poolInfo.penaltyEndBlockNumber = block.number.add(
                penaltyBlockLength
            );

            if(user.amount.add(_amountToStake) > limitForPrize){
                user.onlyPrize = false;
                user.stakeStartDate = block.timestamp;
            }else{
                user.onlyPrize = true;
            }
            user.stakeStatus = true;
            user.userAddress = msg.sender;
            user.userPoolIds.push(randomPoolId);
            user.amount = user.amount.add(_amountToStake);
            allStakedAmount = allStakedAmount.add(_amountToStake);

        }

        allRewardDebt = allRewardDebt.sub(user.rewardDebt);
        user.rewardDebt = user.amount.mul(accTokensPerShare).div(1e18);
        allRewardDebt = allRewardDebt.add(user.rewardDebt);
        emit TokensStaked(msg.sender, _amountToStake, pending);
    }
```

Listing 3.1: `MainPlayPadContract::stakeTokens()`

Recommendation Update the `user.stakeStartDate` for a staker only when the total staked amount of this staker is above the vesting limit for the first time..

Status This issue has been fixed in the following commit: `0fa5e58`.

3.2 Improved Logic In MainPlayPadContract::withdrawPoolRemainder()

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: MainPlayPadContract
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The MainPlayPadContract contract provides a privileged function for the contract approver to withdraw the remaining rewardToken from the staking pool. While examining the withdrawPoolRemainder routine of the MainPlayPadContract contract, we notice the current implementation logic can be improved.

To elaborate, we show below its code snippet. It comes to our attention that this routine can be called by the contract approver at any time. If this routine is called by the approver before finishBlock, the stakers may receive less rewards than they deserve.

```
    function withdrawPoolRemainder() external onlyApprover nonReentrant {
        updatePool();
        uint256 pending =
            allStakedAmount.mul(accTokensPerShare).div(1e18).sub(allRewardDebt);
        uint256 returnAmount = poolTokenAmount.sub(allPaidReward).sub(pending);
        allPaidReward = allPaidReward.add(returnAmount);

        rewardToken.safeTransfer(msg.sender, returnAmount);
        emit WithdrawPoolRemainder(msg.sender, returnAmount);
    }
}
```

Listing 3.2: MainPlayPadContract::withdrawPoolRemainder()

Recommendation Only allow the contract approver to call withdrawPoolRemainder when block.number > finishBlock.

Status

3.3 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MainPlayPadContract`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [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 the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

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

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

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.3: ZRX.sol

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

In current implementation, if we examine the `MainPlayPadContract::withdrawStake()` routine that is designed to withdraw `stakingToken` from the pool for the stakers. To accommodate the specific idiosyncrasy, there is a need to user `safeTransfer()`, instead of `transfer()` (line 1115).

```
1092     // Leave the pool. Claim back your tokens.
1093     // Unclocks the staked + gained tokens and burns pool shares
1094     function withdrawStake(uint256 _amount, uint256 _poolId)
1095         external
1096         nonReentrant
1097     {
1098         UserInfo storage user = userInfo[msg.sender];
1099         UserPoolInfo storage poolInfo = userPoolInfo[_poolId];
1100         require(user.amount >= _amount, "withdraw: not good");
1101         require(poolInfo.amount >= _amount, "withdraw: not good");
1102         require(poolInfo.owner == msg.sender, "you are not owner");
1103         updatePool();
1104         uint256 pending = transferPendingReward(user);
1105         uint256 penaltyAmount = 0;
1106 
1107         if (_amount > 0) {
1108             user.amount = user.amount.sub(_amount);
1109             poolInfo.amount = poolInfo.amount.sub(_amount);
1110 
1111             if (isPenalty) {
1112                 if (block.number < finishBlock) {
1113                     if (block.number <= poolInfo.penaltyEndBlockNumber) {
1114                         penaltyAmount = penaltyRate.mul(_amount).div(1e6);
1115                         stakingToken.transfer(penaltyAddress, penaltyAmount);
1116                     }
1117                 }
1118             }
1119 
1120             stakingToken.safeTransfer(msg.sender, _amount.sub(penaltyAmount));
1121             if (user.amount == 0) {
1122                 participants -= 1;
1123                 user.onlyPrize = true;
1124                 user.stakeStartDate = 0;
1125             }
1126 
1127             if(user.amount > limitForPrize){
1128                 user.onlyPrize = false;
1129             }else{
1130                 user.onlyPrize = true;
1131                 user.stakeStartDate = 0;
1132             }
```

```
        }



        allRewardDebt = allRewardDebt.sub(user.rewardDebt);
        user.rewardDebt = user.amount.mul(accTokensPerShare).div(1e18);
        allRewardDebt = allRewardDebt.add(user.rewardDebt);
        allStakedAmount = allStakedAmount.sub(_amount);

        emit StakeWithdrawn(msg.sender, _amount, pending);
    }
```

Listing 3.4: MainPlayPadContract::withdrawStake()

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

Status This issue has been confirmed.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: MainPlayPadContract/PlayPadIdoContract
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the PlayPad-IDO-DQ protocol, there exist certain privileged accounts that play critical roles in governing and regulating the system-wide operations. In the following, we examine these privileged accounts and their related privileged accesses in current contracts.

Firstly, the privileged functions in the MainPlayPadContract contract allow the the approver to create new IDO contract, change the limitForPrize, and withdraw the remaining rewardToken from the staking pool.

```
    // creates new IDO contract following datas as below
    function createIDO(
        IERC20 _busdAddress,
        IERC20 _saleToken,
        bool _contractStatus,
        uint256 _hardcapUsd,
        uint256 _totalSellAmountToken,
```

```
        uint256 _maxInvestorCount,
        uint256 _maxBuyValue,
        uint256 _minBuyValue,
        uint256 _startTime,
        uint256 _endTime
    ) external nonReentrant onlyApprover {
         PlayPadIdoContract newIdoContract = new PlayPadIdoContract(
            _busdAddress,
            _saleToken,
            _contractStatus,
            _hardcapUsd,
            _totalSellAmountToken,
            _maxInvestorCount,
            _maxBuyValue,
            _minBuyValue,
            _startTime,
            _endTime
        );
        newIdo.push(address(newIdoContract)); // Adding All IDOs
        newIdoContract.transferOwnership(msg.sender);
        isIdoDeployed[address(newIdoContract)] = true;
        emit NewIdoCreated(address(newIdoContract));
    }

    function changePrizeLimit(uint256 _amountToStake) external nonReentrant onlyApprover
         {
        limitForPrize = _amountToStake;
    }
```

Listing 3.5: `MainPlayPadContract::createIDO()/changePrizeLimit()`

```
    function withdrawPoolRemainder() external onlyApprover nonReentrant {
        updatePool();
        uint256 pending =
            allStakedAmount.mul(accTokensPerShare).div(1e18).sub(allRewardDebt);
        uint256 returnAmount = poolTokenAmount.sub(allPaidReward).sub(pending);
        allPaidReward = allPaidReward.add(returnAmount);

        rewardToken.safeTransfer(msg.sender, returnAmount);
        emit WithdrawPoolRemainder(msg.sender, returnAmount);
    }
```

Listing 3.6: `MainPlayPadContract::withdrawPoolRemainder()`

Secondly, the privileged functions in the `TokenReward` contract allow the `owner` to configure key parameters for the contract. These parameters include `contractStatus`, `saleToken`, `lockTime`, `maxBuyValue`, and `minBuyValue`.

```
    // function to change status of contract
    function changePause(bool _contractStatus) public onlyOwner nonReentrant{
        contractStatus = _contractStatus;
    }
```

```

     function changeSaleTokenAddress(IERC20 _contractAddress) external onlyOwner
        nonReentrant {
        saleToken = _contractAddress;
    }
```

Listing 3.7: `PlayPadIdoContract::changePause()/changeSaleTokenAddress()`

```
    //change lock time to prevent missing values
    function changeLockTime(uint256 _lockTime) external nonReentrant onlyOwner {
        lockTime = _lockTime;
    }
```

Listing 3.8: `PlayPadIdoContract::changeLockTime()`

```
    function changeMaxMinBuyLimit(uint256 _maxBuyLimit, uint256 _minBuyLimit) external
        onlyOwner nonReentrant {
        maxBuyValue = _maxBuyLimit;
        minBuyValue = _minBuyLimit;
    }
```

Listing 3.9: `PlayPadIdoContract::changeMaxMinBuyLimit()`

Thirdly, the `emergencyWithdrawAllBusd()` and `withdrawTokens()` functions in the `PlayPadIdoContract` contract allow the `owner` to withdraw all the `busdToken` and `saleToken` held by the contract.

```
    //emergency withdraw function in worst cases
    function emergencyWithdrawAllBusd() external nonReentrant onlyOwner {
        require(busdToken.transferFrom(address(this), msg.sender, busdToken.balanceOf(
            address(this))));
    }
```

Listing 3.10: `PlayPadIdoContract::emergencyWithdrawAllBusd()`

```
    //emergency withdraw for tokens in worst cases
     function withdrawTokens() external nonReentrant onlyOwner {
        require(saleToken.transfer(msg.sender, saleToken.balanceOf(address(this))));
    }
```

Listing 3.11: `PlayPadIdoContract::withdrawTokens()`

Lastly, the `addNewClaimRound()` and `addUsersToWhitelist()` functions in the `PlayPadIdoContract` contract allow the `owner` to add new `claim round` and whitelist the user.

```
    //add new claim round
    function addNewClaimRound(uint256 _roundNumber, uint256 _roundStartDate, uint256
        _claimPercent) external nonReentrant onlyOwner {
        roundDatas storage roundDetail = _roundDatas[_roundNumber];
        roundDetail.roundStartDate = _roundStartDate;
        roundDetail.roundPercent = _claimPercent;
        claimRoundsDate.push(_roundStartDate);
    }
```

Listing 3.12: `PlayPadIdoContract::addNewClaimRound()`

```
    function addUsersToWhitelist(address[] memory _whitelistedAddresses, uint256[]
        memory _buyingLimits) external onlyOwner nonReentrant{
        for(uint256 i = 0;  i < _whitelistedAddresses.length; i++){
             whitelistedInvestorData storage investor = _investorData[
                 _whitelistedAddresses[i]];
             investor.totalVesting = _buyingLimits[i];
             investor.isWhitelisted = true;
             whitelistedAddresses.push(_whitelistedAddresses[i]);
        }

    }
```

Listing 3.13: `PlayPadIdoContract::addUsersToWhitelist()`

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

Recommendation Make the list of extra privileges granted to `approver/owner` explicit to `PlayPad-IDO-DQ` protocol users.

Status This issue has been confirmed. The team confirms that `DAO` will be used in the future to solve this trust issue of admin keys by `multi-sig`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `PlayPad-IDO-DQ` protocol.The `PlayPad DQ` moves away from the uniform standardized model for `IGO` participation and allows investors to get vestings with less stakes. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

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

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

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

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

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