SMART CONTRACT AUDIT REPORT

for

Oddz Finance

Prepared By: Shuxiao Wang

PeckShield

June 18, 2021

Document Properties

Client	Oddz Finance
Title	Smart Contract Audit Report
Target	Oddz
Version	1.0
Author	Xuxian Jiang
Auditors	Stephen Bie, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 18, 2021	Xuxian Jiang	Final Release
1.0-rc1	June 4, 2021	Xuxian Jiang	Release Candidate #1
0.4	June 3, 2021	Xuxian Jiang	Add More Findings #3
0.3	June 1, 2021	Xuxian Jiang	Add More Findings #2
0.2	May 28, 2021	Xuxian Jiang	Add More Findings #1
0.1	May 14, 2021	Xuxian Jiang	Initial Draft

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

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

1.1 About Oddz

`Oddz` is a multi-chain options trading platform on `Binance Smart Chain`, `Polkadot` and `Ethereum` to allow users to trade customized options with rewards. It has a built-in oracle solution with distinguishing features like instant trades, interoperable option trades, transparent premium discovery mechanisms, and customized option trading techniques. `Oddz` provides a valuable instrument to hedge risks and control excessive exposure from market fluctuation and dynamics, therefore presenting a unique contribution to current DeFi ecosystem.

The basic information of Oddz is as follows:

Table 1.1: Basic Information of Oddz

Item	Description
Issuer	Oddz Finance
Website	https://oddz.fi/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 18, 2021

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

- https://github.com/oddz-finance/oddz-contracts.git (dbc8506)

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

- https://github.com/oddz-finance/oddz-contracts.git (c95e16e)

1.2 About PeckShield

PeckShield Inc. [11] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [10]:

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

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

Table 1.3: The Full Audit Checklist

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

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

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

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [9], 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.

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 Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

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

Severity	# of Findings	
Critical	0	
High	2	■■
Medium	2	■■
Low	4	■■■■
Informational	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 need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 high-severity vulnerabilities, 2 medium-severity vulnerabilities, 4 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Oddz Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Support of Non-Compliant ERC20 Tokens	Coding Practices	Fixed
PVE-002	Low	Possible Sandwich/MEV Attacks For Reduced Returns	Time and State	Fixed
PVE-003	High	Possible Contamination Of daysActiveLiquidity Records	Business Logic	Fixed
PVE-004	Low	Improper Premium Distribution	Business Logic	Fixed
PVE-005	Medium	Proper Administrator Allowance Management	Coding Practices	Fixed
PVE-006	Low	Possible Front-running/MEV For Maximum Discount	Time and State	Mitigated
PVE-007	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-008	Informational	Improved Validation Of Function Arguments	Coding Practices	Fixed
PVE-009	High	Improved Logic In _stake()/_unstake()	Business Logic	Fixed

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 Support of Non-Compliant ERC20 Tokens

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PancakeSwapForUnderlyingAsset`
- 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 `approve()` routine and analyze 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. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
/**
* @dev Approve the passed address to spend the specified amount of tokens on behalf
    of msg.sender.
* @param _spender The address which will spend the funds.
* @param _value The amount of tokens to be spent.
*/
function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

    // To change the approve amount you first have to reduce the addresses`
    //  allowance to zero by calling `approve(_spender, 0)` if it is not
    //  already 0 to mitigate the race condition described here:
    //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
    require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));
```

```
207         allowed[msg.sender][_spender] = _value;
208         Approval(msg.sender, _spender, _value);
209     }
```

Listing 3.1: USDT Token Contract

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. In the following, we use the `PancakeSwapForUnderlyingAsset::swapTokensForUA()` routine as an example. This routine is designed to approve a specific token for swap contract. To accommodate the specific idiosyncrasy, there is a need to `approve()` twice (lines $200 - 201$): the first one reduces the allowance to 0; and the second one sets the new allowance.

```
30      function swapTokensForUA(
31          address _fromToken,
32          address _toToken,
33          address _account,
34          uint256 _amountIn,
35          uint256 _deadline
36      ) public override onlyOwner returns (uint256[] memory result) {
37          address[] memory path = new address[](2);
38          path[0] = _fromToken;
39          path[1] = _toToken;
40          ERC20(_fromToken).approve(address(pancakeSwap), _amountIn);
41          result = pancakeSwap.swapExactTokensForTokens(_amountIn, amountOutMin, path,
                address(this), _deadline);
42          // converting address to address payable
43          ERC20(address(uint160(_toToken))).safeTransfer(_account, result[1]);
44      }
```

Listing 3.2: PancakeSwapForUnderlyingAsset::swapTokensForUA()

Moreover, it is important to note that for certain non-compliant ERC20 tokens (e.g., USDT), the `transfer()` function does not have a return value. 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 will be unfortunately reverted.

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

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

Status The issue has been fixed by this commit: `e16d427`.

3.2 Possible Sandwich/MEV Attacks For Reduced Returns

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PancakeSwapForUnderlyingAsset`
- Category: Time and State [8]
- CWE subcategory: CWE-682 [3]

Description

As mentioned in Section 3.1, the `PancakeSwapForUnderlyingAsset` contract has a helper routine, i.e., `swapTokensForUA()`, that is designed to swap tokens. It has a rather straightforward logic in allowing `pancakeSwap` to transfer the funds and then calling `swapExactTokensForTokens()` to actually perform the intended token swap.

```
    function swapTokensForUA(
        address _fromToken,
        address _toToken,
        address _account,
        uint256 _amountIn,
        uint256 _deadline
    ) public override onlyOwner returns (uint256[] memory result) {
        address[] memory path = new address[](2);
        path[0] = _fromToken;
        path[1] = _toToken;
        ERC20(_fromToken).approve(address(pancakeSwap), _amountIn);
        result = pancakeSwap.swapExactTokensForTokens(_amountIn, amountOutMin, path,
            address(this), _deadline);
        // converting address to address payable
        ERC20(address(uint160(_toToken))).safeTransfer(_account, result[1]);
    }
```

Listing 3.3: PancakeSwapForUnderlyingAsset::swapTokensForUA()

To elaborate, we show above the `swapTokensForUA()` routine. We notice the token swap is routed to `pancakeSwap` and the actual swap operation `swapExactTokensForTokens()` essentially does not specify any restriction (with `amountOutMin=0`) on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of yielding.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation to the above front-running attack to better protect the interests of farming users.

Status The issue has been fixed by this commit: `7b6baef`.

3.3 Possible Contamination Of daysActiveLiquidity Records

- ID: PVE-003
- Severity: High
- Likelihood: Medium
- Impact:High
- Target: `AbstractOddzPool`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `Oddz` protocol is a multi-chain peer-to-pool options trading protocol. For each option being purchased, the pool will lock certain amount of funds to meet the need in case the option will be exercised (at the agreed strike price) within the option's validity period. Internally, a storage variable named `lockedAmount` keeps track of total locked amount of funds in current pool for active options. Also, it uses another storage variable `daysActiveLiquidity` to record the active liquidity on a daily basis.

To elaborate, we show below the `getDaysActiveLiquidity()` routine that allows for the retrieval of active liquidity for a date. It is a public function and any one is able to call it. It comes to our attention this public function is not a `view` function. Instead, it allows for the update of `daysActiveLiquidity` by simply assuming current liquidity (line 308). With that, a malicious actor may pre-populate the active liquidity for a future date, which may completely mess up the internal accounting of active liquidity for a date!

```
299     /**
300      * @notice Get active liquidity for a date
301      * @param _date liquidity date
302      */
303     function getDaysActiveLiquidity(uint256 _date) public override returns (uint256
            _liquidity) {
304         // Skip for the first time liqiduity
305         if (daysActiveLiquidity[_date] == 0 && latestLiquidityEvent != 0) {
306             uint256 stDate = latestLiquidityEvent;
307             while (stDate <= _date) {
308                 daysActiveLiquidity[stDate] = daysActiveLiquidity[latestLiquidityEvent];
309                 stDate = stDate + 1 days;
```

```
            }
        }
        _liquidity = daysActiveLiquidity[_date];
    }
```

Listing 3.4: AbstractOddzPool::getDaysActiveLiquidity()

Note another routine `AbstractTokenStaking::getAndUpdateDaysActiveStake()` shares the same issue.

Recommendation Prevent the above accounting information from being contaminated.

Status The issue has been fixed by this commit: `a0108a8`.

3.4 Improper Premium Distribution

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `AbstractOddzPool`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

As mentioned in Section 3.3, the `Oddz` protocol is a multi-chain peer-to-pool options trading protocol. To support the pool-based options trading, the `Oddz` protocol has an integrated `AbstractOddzPool` contract. This pool contract allows for liquidity addition and removal as well as options premium dissemination. While examining the premium dissemination logic, we notice an issue that mis-calculates the eligible premium for dissemination.

To elaborate, we show below the `enablePremiumDistribution()` routine that, as the name indicates, enables premium distribution for a specific date. However, when the premium for a specific date is negative (the `else`-branch at lines 205 – 208), the current computation for `eligible` incorrectly adds `surplus`, which instead needs to be subtracted. In other words, the proper calculation should be the following `premium.eligible = daysExercise[_date] - premium.collected - surplus;`.

```
193     /**
194      * @notice Enable premium distribution for a date
195      * @param _date Premium eligibility date
196      */
197     function enablePremiumDistribution(uint256 _date) public override {
198         require(_date < DateTimeLibrary.getPresentDayTimestamp(), "LP Error: Invalid
                Date");
199         PremiumPool storage premium = premiumDayPool[_date];
200         require(!premium.enabled, "LP Error: Premium eligibilty already updated for the
                date");
201         premium.enabled = true;
```

```
        if (premium.collected + surplus >= daysExercise[_date]) {
            premium.eligible = premium.collected + surplus - daysExercise[_date];
            premium.isNegative = false;
        } else {
            premium.eligible = daysExercise[_date] - premium.collected + surplus;
            premium.isNegative = true;
        }
        surplus = 0;
    }
```

Listing 3.5: AbstractOddzPool::enablePremiumDistribution()

Recommendation Revise the affected `enablePremiumDistribution()` routine to compute the right `eligible` amount.

Status The issue has been fixed as the affected function has been removed.

3.5 Proper Administrator Allowance Management

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `OddzOptionManager`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

The `Oddz` protocol has a protocol-wide `OddzAdministrator` contract that can be used to configure various aspects of the protocol. And the options management contract `OddzOptionManager` has a permissioned function `setAdministrator()` that allows for the reconfiguration of a new `Oddz` administrator address.

To elaborate, we show below the `setAdministrator()` routine. This routine not only sets up the new administrator, but also permits the new administer contract to transfer the funds locked in the options management contract. We notice there is also a need to reset the spending allowance of the old administrator back to 0.

```
    /**
     * @notice sets administrator address
     * @param _administrator Oddz administrator address
     */
    function setAdministrator(IOddzAdministrator _administrator) external onlyOwner {
        require(address(_administrator).isContract(), "invalid SDK contract address");
        administrator = _administrator;

        // Approve token transfer to administrator contract
```

```
        token.approve(address(administrator), type(uint256).max);
    }
```

Listing 3.6: OddzOptionManager::setAdministrator()

Recommendation Reset the allowance of the old administrator, if any, back to 0

Status The issue has been fixed by this commit: `fe0f2b2`.

3.6 Possible Front-running/MEV For Maximum Discount

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Time and State [8]
- CWE subcategory: CWE-682 [3]

Description

As part of the incentive mechanisms, the `Oddz` protocol offers certain discounts on the options-related transaction fee and settlement fee. Based on the supported tokens, the actual discount for an option buyer may vary with the holding balance. If a user has a larger balance, the user may be offered a larger discount. Our analysis shows this mechanism may be abused to always obtain maximum discount.

To elaborate, we show below the `getTransactionFee()` routine. As the name indicates, the function calculates the intended transaction fee for an option buyer. Note that the discount is largely affected by the computed `numDigits(txnFeeTokens[i].balanceOf(_buyer)` (line 90), which may be leveraged to have a flashloan to ensure the current option buyer can always obtain the maximum discount.

```
78      /**
79       * @notice Gets transaction fee for an option buyer
80       * @param _buyer Address of buyer
81       * @return txnFee Transaction fee percentage for the buyer
82       */
83      function getTransactionFee(address _buyer) public view override returns (uint256
            txnFee) {
84          uint256 maxDiscount;
85          txnFee = txnFeePerc * 10**decimals;
86          for (uint256 i = 0; i < txnFeeTokens.length; i++) {
87              if (txnFeeTokens[i].balanceOf(_buyer) == 0) continue;
88              uint256 discount =
89                  tokenFeeDiscounts[txnFeeTokens[i]][
90                      numDigits(txnFeeTokens[i].balanceOf(_buyer) / 10**txnFeeTokens[i].
                            decimals())
91                  ];
```

```
            if (discount > maxDiscount) maxDiscount = discount;
        }
        txnFee -= (txnFeePerc * maxDiscount * 10**decimals) / 100;
    }
```

Listing 3.7: OddzFeeManager::getTransactionFee()

Recommendation Improve the above discount mechanism to ensure the user tokens are locked when the balance is used for discount calculation.

Status The issue has been confirmed. By design, the protocol will only support staking tokens which have a minimum locking period of 7 days.

3.7 Trust Issue of Admin Keys

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: OddzStakingManager
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the Oddz protocol, there is 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 representative privileged operations in the Oddz's OddzStakingManager contract.

```
    /**
     * @notice Set lockup duration for the token
     * @param _token token address
     * @param _duration lockup duration
     */
    function setLockupDuration(address _token, uint256 _duration)
        external
        onlyOwner
        validToken(_token)
        validDuration(_duration)
    {
        tokens[_token]._lockupDuration = _duration;
    }

    /**
     * @notice Deactivate token
     * @param _token token address
     */
    function deactivateToken(address _token) external onlyOwner validToken(_token) {
        tokens[_token]._active = false;
```

```
        emit TokenDeactivate(_token, tokens[_token]._name);
    }

    /**
     * @notice Activate token
     * @param _token token address
     */
    function activateToken(address _token) external onlyOwner inactiveToken(_token) {
        tokens[_token]._active = true;
        emit TokenActivate(_token, tokens[_token]._name);
    }
```

Listing 3.8: Example Privileged Operations in OddzStakingManager

We emphasize that the privilege assignment is necessary and consistent with the token design. However, it is worrisome if the `owner` is not governed by a `DAO`-like structure. The discussion with the team has confirmed that this privileged account will be managed by a multi-sig account. Note that a compromised `owner` account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the `Oddz` protocol.

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 This issue has been confirmed and partially mitigated with the following commit: `95b4b5a`.

3.8 Improved Validation Of Function Arguments

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

The `Oddz` protocol supports more than a dozen of pools and these pools are managed via a single pool manager contract, i.e., `OddzLiquidityPoolManager`. While examining this pool manager contract, we notice a unique feature that allows the liquidity to be moved between pools.

To elaborate, we show below the `_poolTransfer()()` function inside the pool manager contract. This function allows liquidity providers to move the provided liquidity between pools. However, this function can be improved by ensuring the sending pool and receiving pool are valid. The current implementation only guarantees the validity of the sending pool, not the receiving pool.

```
    /**
     * @notice Move liquidity between pools
     * @param _poolTransfer source and destination pools with amount of transfer
     */
    function move(PoolTransfer memory _poolTransfer) external {
        require(
            lastPoolTransfer[msg.sender] == 0  (lastPoolTransfer[msg.sender] + 1 weeks)
                < block.timestamp,
            "LP Error: Pool transfer available only once in 7 days"
        );
        lastPoolTransfer[msg.sender] = block.timestamp;
        int256 totalTransfer = 0;
        for (uint256 i = 0; i < _poolTransfer._source.length; i++) {
            require(validPools[_poolTransfer._source[i]], "LP Error: Invalid pool");
            _removeLiquidity(_poolTransfer._source[i], _poolTransfer._sAmount[i]);
            totalTransfer += int256(_poolTransfer._sAmount[i]);
        }
        for (uint256 i = 0; i < _poolTransfer._destination.length; i++) {
            _poolTransfer._destination[i].addLiquidity(_poolTransfer._dAmount[i], msg.
                sender);
            totalTransfer -= int256(_poolTransfer._dAmount[i]);
        }
        require(totalTransfer == 0, "LP Error: invalid transfer amount");
    }
```

Listing 3.9: OddzLiquidityPoolManager::_poolTransfer()

Recommendation Apply improved validations on the above `move()` routine.

Status The issue has been fixed by this commit: `c229eb1`.

3.9 Improved Logic In _stake()/_unstake()

- ID: PVE-009
- Severity: High
- Likelihood: High
- Impact:Medium
- Target: `AbstractTokenStaking`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In order to engage protocol users, the `Oddz` protocol has developed a `staking` mechanism to incentivize protocol users. To do that, it naturally supports two staking-related functions `stake()` and `unstake()`.

In the meantime, the protocol imposes certain lockup time for staking users and the lockup time is managed by the system parameter `lockupDuration`.

While examining the staking/unstaking logic, we notice it may cause the permanent loss of previous rewards of a staking user. To elaborate, we show below the related `_stake()` routine. It properly calls `allocateStakerRewards()` to compute current staking records. However, it immediately resets the staking record by completely disregarding the accumulated rewards (line 179). To mitigate, there is a need to restore the accumulated records so that the user may properly reclaim them later. Note the `_unstake()` routine shares the same issue.

```
166     /**
167      * @notice Updates user stake
168      * @param _staker Address of the staker
169      * @param _amount Amount to stake
170      * @param _date  Date on which tokens are staked
171      */
172     function _stake(
173         address _staker,
174         uint256 _amount,
175         uint256 _date
176     ) internal onlyOwner {
177         _allocateStakerRewards(_staker, _date);
178         // update to stake to hold existing stake
179         staker[_staker] = UserStake(staker[_staker]._amount + _amount, _date, 0, 0);
180 
181         dayStakeMap[_date]._totalActiveStake = getAndUpdateDaysActiveStake(_date) + _amount;
182         lastStaked = _date;
183     }
184 
185     /**
186      * @notice updates user unstake tokens
187      * @param _amount Amount to burn and transfer
188      * @param _date  Date on which tokens are unstaked
189      */
190     function _unstake(
191         address _staker,
192         uint256 _amount,
193         uint256 _date
194     ) internal {
195         _allocateStakerRewards(_staker, _date);
196         // update to stake to hold existing stake
197         staker[_staker] = UserStake(staker[_staker]._amount - _amount, _date, 0, 0);
198 
199         dayStakeMap[_date]._totalActiveStake = getAndUpdateDaysActiveStake(_date) - _amount;
200         lastStaked = _date;
201     }
```

Listing 3.10: AbstractTokenStaking::_stake()/_unstake()

Recommendation Revise the above `_stake()/_unstake()` routines to properly save accumulated staking records.

Status The issue has been fixed by this commit: `7bd25de`.

4 Conclusion

In this audit, we have analyzed the Oddz design and implementation. The system presents a unique, robust offering as a decentralized non-custodial multi-chain options trading platform that allows users to trade customized options with rewards. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

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

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

[3] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.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 CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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