SMART CONTRACT AUDIT REPORT

for

Duneswap Protocol

Prepared By: Yiqun Chen

PeckShield

February 26, 2022

Document Properties

Client	Duneswap
Title	Smart Contract Audit Report
Target	Duneswap
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	February 26, 2022	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 source code of the Duneswap 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 Duneswap

Duneswap is an automated market maker or decentralized exchange native to Oasis Emerald ParaTime network. The protocol is forked from the popular UniswapV2 protocol with extensions for customized farming and staking suppport. Protocol users may be rewarded with the protocol token DUNE, which is the native token to the platform and can be attained by farming/staking or trading for. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Duneswap

Item	Description
Name	Duneswap
Website	https://www.duneswap.com/
Type	Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 26, 2022

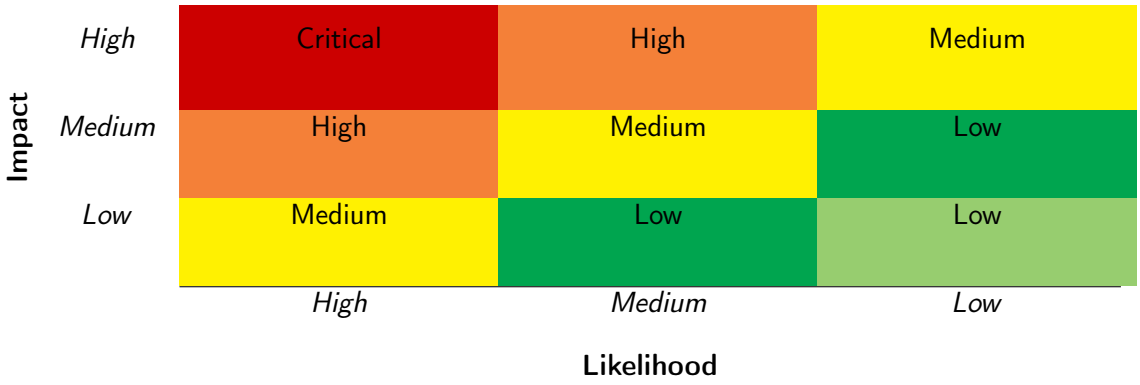
In the following, we show the Git repository of reviewed file and the commit hash value used in this audit:

- https://github.com/duneswap/duneswap-contracts.git (85fd6a8)

1.2 About PeckShield

PeckShield Inc. [9] 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



1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- 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 deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would

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

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

Severity	# of Findings	
Critical	0	
High	1	■
Medium	1	■
Low	4	■■■■
Informational	0	
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 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 vulnerability, and 4 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Incorrect walletTokenBalance Update In DuneLocker::lockTokens()	Business Logic	Fixed
PVE-002	Low	Fork-Resistant Domain Separator In DuneERC20	Business Logic	Confirmed
PVE-003	Low	Implicit Assumption Enforcement In AddLiquidity()	Coding Practices	Fixed
PVE-004	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Confirmed
PVE-005	Low	Duplicate Pool Detection And Prevention	Business Logic	Confirmed
PVE-006	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 Incorrect walletTokenBalance Update In DuneLocker::lockTokens()

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: `DuneLocker`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `DuneLocker` contract provides an external `lockTokens()` function for users to lock a specified amount of `ERC20` tokens into the contract. Only the specified withdrawer can withdraw these locked tokens when the lock time is due. Our analysis with this routine shows its current implementation is not correct.

To elaborate, we show below its code snippet. It comes to our attention that the wrong key is used when updating the state variable `walletTokenBalance`. Specifically, the second key used for updating the nested mapping state variable `walletTokenBalance` should be `_withdrawer`, instead of current `msg.sender` (line 52).

```
37      function lockTokens(IERC20 _token, address _withdrawer, uint256 _amount, uint256
            _unlockTimestamp) payable external returns (uint256 _id) {
38          require(_amount > 0, 'Token amount too low!');
39          require(_unlockTimestamp < 10000000000, 'Unlock timestamp is not in seconds!');
40          require(_unlockTimestamp > block.timestamp, 'Unlock timestamp is not in the
                future!');
41          require(_token.allowance(msg.sender, address(this)) >= _amount, 'Approve tokens
                first!');
42          require(msg.value >= lockFee, 'Need to pay lock fee!');
43
44          uint256 beforeDeposit = _token.balanceOf(address(this));
45          _token.safeTransferFrom(msg.sender, address(this), _amount);
46          uint256 afterDeposit = _token.balanceOf(address(this));
```

```
47
48          _amount = afterDeposit.sub(beforeDeposit);
49
50          payable(feeAddress).transfer(msg.value);
51
52          walletTokenBalance[address(_token)][msg.sender] = walletTokenBalance[address(
                _token)][msg.sender].add(_amount);
53
54          _id = ++depositsCount;
55          lockedToken[_id].token = _token;
56          lockedToken[_id].withdrawer = _withdrawer;
57          lockedToken[_id].amount = _amount;
58          lockedToken[_id].unlockTimestamp = _unlockTimestamp;
59          lockedToken[_id].withdrawn = false;
60
61          depositsByTokenAddress[address(_token)].push(_id);
62          depositsByWithdrawer[_withdrawer].push(_id);
63
64          emit Lock(address(_token), _amount, _id);
65
66          return _id;
67      }
```

Listing 3.1: `DuneLocker::lockTokens()`

Recommendation Use the correct key when updating the state variable `walletTokenBalance`. An example revision is shown as follows:

```
37      function lockTokens(IERC20 _token, address _withdrawer, uint256 _amount, uint256
            _unlockTimestamp) payable external returns (uint256 _id) {
38          require(_amount > 0, 'Token amount too low!');
39          require(_unlockTimestamp < 10000000000, 'Unlock timestamp is not in seconds!');
40          require(_unlockTimestamp > block.timestamp, 'Unlock timestamp is not in the
                future!');
41          require(_token.allowance(msg.sender, address(this)) >= _amount, 'Approve tokens
                first!');
42          require(msg.value >= lockFee, 'Need to pay lock fee!');
43
44          uint256 beforeDeposit = _token.balanceOf(address(this));
45          _token.safeTransferFrom(msg.sender, address(this), _amount);
46          uint256 afterDeposit = _token.balanceOf(address(this));
47
48          _amount = afterDeposit.sub(beforeDeposit);
49
50          payable(feeAddress).transfer(msg.value);
51
52          walletTokenBalance[address(_token)][_withdrawer] = walletTokenBalance[address(
                _token)][_withdrawer].add(_amount);
53
54          _id = ++depositsCount;
55          lockedToken[_id].token = _token;
56          lockedToken[_id].withdrawer = _withdrawer;
```

```
57          lockedToken[_id].amount = _amount;
58          lockedToken[_id].unlockTimestamp = _unlockTimestamp;
59          lockedToken[_id].withdrawn = false;
60
61          depositsByTokenAddress[address(_token)].push(_id);
62          depositsByWithdrawer[_withdrawer].push(_id);
63
64          emit Lock(address(_token), _amount, _id);
65
66          return _id;
67      }
```

Listing 3.2: DuneLocker::lockTokens()

Status This issue has been fixed in the following commit: d602b5d.

3.2 Fork-Resistant Domain Separator In DuneERC20

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: DuneERC20
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

In the DuneERC20 contract, the state variable DOMAIN_SEPARATOR is immutable because it is only assigned in the constructor() function (lines 34-44).

```
29      constructor() public {
30          uint256 chainId;
31          assembly {
32              chainId := chainid()
33          }
34          DOMAIN_SEPARATOR = keccak256(
35              abi.encode(
36                  keccak256(
37                      "EIP712Domain(string name,string version,uint256 chainId,address
                            verifyingContract)"
38                  ),
39                  keccak256(bytes(name)),
40                  keccak256(bytes("1")),
41                  chainId,
42                  address(this)
43              )
44          );
45      }
```

Listing 3.3: DuneERC20::constructor()

The DOMAIN_SEPARATOR is used in the ERC20-enhancing function, i.e., permit(), which allows for approvals to be made via secp256k1 signatures. When analyzing this permit() routine, we notice the current implementation can be improved by recalculating the value of DOMAIN_SEPARATOR in permit() function. Suppose there is a hard-fork, since DOMAIN_SEPARATOR is immutable, a valid signature for one chain could be replayed on the other.

```
106     function permit(
107         address owner,
108         address spender,
109         uint256 value,
110         uint256 deadline,
111         uint8 v,
112         bytes32 r,
113         bytes32 s
114     ) external {
115         require(deadline >= block.timestamp, "Duneswap: EXPIRED");
116         bytes32 digest = keccak256(
117             abi.encodePacked(
118                 "\x19\x01",
119                 DOMAIN_SEPARATOR,
120                 keccak256(
121                     abi.encode(
122                         PERMIT_TYPEHASH,
123                         owner,
124                         spender,
125                         value,
126                         nonces[owner]++,
127                         deadline
128                     )
129                 )
130             )
131         );
132         address recoveredAddress = ecrecover(digest, v, r, s);
133         require(
134             recoveredAddress != address(0) && recoveredAddress == owner,
135             "Duneswap: INVALID_SIGNATURE"
136         );
137         _approve(owner, spender, value);
138     }
```

Listing 3.4: DuneERC20::permit()

Recommendation Recalculate the value of DOMAIN_SEPARATOR inside the permit() function.

Status This issue has been confirmed.

3.3 Implicit Assumption Enforcement In AddLiquidity()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `DuneRouter02`
- Category: Coding Practices [5]
- CWE subcategory: CWE-628 [2]

Description

In the `DuneRouter02` contract, the `addLiquidity()` routine (see the code snippet below) is provided to add `amountADesired` amount of `tokenA` and `amountBDesired` amount of `tokenB` into the pool as liquidity via the `UniswapRouterV2::addLiquidity()` routine. To elaborate, we show below the related code snippet.

```
80      function addLiquidity(
81          address tokenA,
82          address tokenB,
83          uint256 amountADesired,
84          uint256 amountBDesired,
85          uint256 amountAMin,
86          uint256 amountBMin,
87          address to,
88          uint256 deadline
89      )
90          external
91          virtual
92          override
93          ensure(deadline)
94          returns (
95              uint256 amountA,
96              uint256 amountB,
97              uint256 liquidity
98          )
99      {
100         (amountA, amountB) = _addLiquidity(
101             tokenA,
102             tokenB,
103             amountADesired,
104             amountBDesired,
105             amountAMin,
106             amountBMin
107         );
108         address pair = DuneLibrary.pairFor(factory, tokenA, tokenB);
109         TransferHelper.safeTransferFrom(tokenA, msg.sender, pair, amountA);
110         TransferHelper.safeTransferFrom(tokenB, msg.sender, pair, amountB);
111         liquidity = IDunePair(pair).mint(to);
```

```
112     }
```

Listing 3.5: DuneRouter02::addLiquidity()

```
33      function _addLiquidity(
34          address tokenA,
35          address tokenB,
36          uint256 amountADesired,
37          uint256 amountBDesired,
38          uint256 amountAMin,
39          uint256 amountBMin
40      ) internal virtual returns (uint256 amountA, uint256 amountB) {
41          // create the pair if it doesn't exist yet
42          if (IDuneFactory(factory).getPair(tokenA, tokenB) == address(0)) {
43              IDuneFactory(factory).createPair(tokenA, tokenB);
44          }
45          (uint256 reserveA, uint256 reserveB) = DuneLibrary.getReserves(
46              factory,
47              tokenA,
48              tokenB
49          );
50          if (reserveA == 0 && reserveB == 0) {
51              (amountA, amountB) = (amountADesired, amountBDesired);
52          } else {
53              uint256 amountBOptimal = DuneLibrary.quote(
54                  amountADesired,
55                  reserveA,
56                  reserveB
57              );
58              if (amountBOptimal <= amountBDesired) {
59                  require(
60                      amountBOptimal >= amountBMin,
61                      "DuneRouter: INSUFFICIENT_B_AMOUNT"
62                  );
63                  (amountA, amountB) = (amountADesired, amountBOptimal);
64              } else {
65                  uint256 amountAOptimal = DuneLibrary.quote(
66                      amountBDesired,
67                      reserveB,
68                      reserveA
69                  );
70                  assert(amountAOptimal <= amountADesired);
71                  require(
72                      amountAOptimal >= amountAMin,
73                      "DuneRouter: INSUFFICIENT_A_AMOUNT"
74                  );
75                  (amountA, amountB) = (amountAOptimal, amountBDesired);
76              }
77          }
78      }
```

Listing 3.6: DuneRouter02::_addLiquidity()

It comes to our attention that the DuneRouter02 has implicit assumptions on the _addLiquidity() routine. The above routine takes two amounts: amountXDesired and amountXMin. The first amount amountXDesired determines the desired amount for adding liquidity to the pool and the second amount amountXMin determines the minimum amount of used assets. There are two implicit conditions, i.e., amountADesired >= amountAMin and amountBDesired >= amountBMin. However, if these two conditions are not met, current logic will not trigger reverts because the code above performs asymmetric checks for these amounts. Hence, without stating these assumptions, slippage control for some trades on UniswapV2 Router may not be checked and may not be taken into account at all in certain scenarios.

Recommendation Make the requirement of amountADesired >= amountAMin and amountBDesired >= amountBMin explicitly in the addLiquidity() function.

Status This issue has been fixed in the following commit: d602b5d.

3.4 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: DuneswapToken
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

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."*

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

```
68              balances[_to] += _value;
69              Transfer(msg.sender, _to, _value);
70              return true;
71          } else { return false; }
72      }

74      function transferFrom(address _from, address _to, uint _value) returns (bool) {
75          if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
                balances[_to] + _value >= balances[_to]) {
76              balances[_to] += _value;
77              balances[_from] -= _value;
78              allowed[_from][msg.sender] -= _value;
79              Transfer(_from, _to, _value);
80              return true;
81          } else { return false; }
82      }
```

Listing 3.7: 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 `DuneswapToken::rescueTokens()` routine that is designed to rescue `token` from the contract. To accommodate the specific idiosyncrasy, there is a need to user `safeTransfer()`, instead of `transfer()` (line 68).

```
67      function rescueTokens(IERC20 token, uint256 value) external onlyRole(RESCUER_ROLE) {
68          token.transfer(_msgSender(), value);
69
70          emit TokensRescued(_msgSender(), address(token), value);
71      }
```

Listing 3.8: `MainPlayPadContract::rescueTokens()`

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

Status This issue has been confirmed.

3.5 Duplicate Pool Detection And Prevention

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `DuneDistributor/DuneVault`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `Duneswap` protocol allows the `owner` to initialize the liquidity mining pool by calling the `add()` function. In this function, the `owner` is able to add the liquidity token (e.g., `lpToken`), and set the mining distribution rule (i.e., `allocPoint`) for the pool. However, it doesn't consider the situation that the `owner` may add the same liquidity token more than once. To elaborate, we list below the `add()` function.

```
171     // Add a new lp to the pool. Can only be called by the owner.
172     // Can add multiple pool with same lp token without messing up rewards, because each
            pool's balance is tracked using its own totalLp
173     function add(
174         uint256 _allocPoint,
175         IERC20 _lpToken,
176         uint16 _depositFeeBP,
177         uint256 _harvestInterval
178     ) public onlyOwner {
179         require(
180             _depositFeeBP <= MAXIMUM_DEPOSIT_FEE_RATE,
181             "add: deposit fee too high"
182         );
183         require(
184             _harvestInterval <= MAXIMUM_HARVEST_INTERVAL,
185             "add: invalid harvest interval"
186         );
187         massUpdatePools();
188         uint256 lastRewardBlock = block.number > startBlock
189             ? block.number
190             : startBlock;
191         totalAllocPoint = totalAllocPoint.add(_allocPoint);
192         poolInfo.push(
193             PoolInfo({
194                 lpToken: _lpToken,
195                 allocPoint: _allocPoint,
196                 lastRewardBlock: lastRewardBlock,
197                 accDunePerShare: 0,
198                 depositFeeBP: _depositFeeBP,
199                 harvestInterval: _harvestInterval,
200                 totalLp: 0
201             })
```

```
202         );
203     }
```

Listing 3.9: DuneDistributor::add()

As shown in the above implementation, if the owner adds the same liquidity token more than once, rewards will be messed up. With that, we suggest to add necessary validation to ensure the same liquidity token will not be added twice. Note the DuneVault::add() routine shares a similar issue.

Recommendation Revise the above add() routine to better validate the owner input to ensure no duplicate pool will be created.

Status This issue has been confirmed.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: Multiple contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the Duneswap protocol, there are six privileged accounts, i.e., owner, PAUSER_ROLE, MINTER_ROLE, RESCUER_ROLE, _operator, and feeToSetter. These accounts play a critical role in governing and regulating the protocol-wide operations (e.g., pause/unpause the DuneswapToken contract, mint Duneswap tokens for a specified account, rescue tokens sent to the DuneswapToken contract by mistake, update a given pool's Dune allocation point and deposit fee, add a new LP to the farm pool, update the Dune distribution rate per block, and set the key parameters, etc.).

In the following, we use the DuneswapToken contract as an example and show the representative functions potentially affected by the privileges of the PAUSER_ROLE/MINTER_ROLE accounts.

```
46      function pause() public onlyRole(PAUSER_ROLE) {
47          _pause();
48      }
49
50      function unpause() public onlyRole(PAUSER_ROLE) {
51          _unpause();
52      }
53
54      function mint(address to, uint256 amount) public onlyRole(MINTER_ROLE) {
```

```
55          require(totalSupply() + amount <= _maxSupply, "ERC20: cannot mint more tokens,
                cap exceeded");
56          _mint(to, amount);
57      }
58
59      function _beforeTokenTransfer(
60          address from,
61          address to,
62          uint256 amount
63      ) internal override whenNotPaused {
64          super._beforeTokenTransfer(from, to, amount);
65      }
```

Listing 3.10: DuneswapToken::pause()/unpause()/mint()

The first function pause() allows for the PAUSER_ROLE to pause the DuneswapToken contract. The second function unpause() allows for the PAUSER_ROLE to unpause the DuneswapToken contract. And the third function mint() allows for the MINTER_ROLE to mint more tokens into circulation for a specified account. We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to the 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 privileged accounts explicit to Duneswap token users

Status This issue has been confirmed. The team confirms that there is a timelock contract in place for Duneswap protocol.

4 Conclusion

In this audit, we have analyzed the design and implementation of the Duneswap protocol. Duneswap is an Automated Market Maker/Decentralized Exchange native to Oasis Emerald ParaTime Network. 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-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.html.

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

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

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

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

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

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

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