Security Audit Report for Ref DCL Contract

Date: September 28th, 2023

Version: 4.0

Contact: contact@blocksec.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2023.10.07 10:28:22 +08'00'

Contents

Report Manifest

Item	Description
Client	Ref-Finance
Target	Ref DCL Contract

Version History

Version	Date	Description
1.0	December 5th, 2022	First Release
2.0	February 10th, 2023	Second Release
3.0	July 10th, 2023	Third Release
4.0	September 28th, 2023	Fourth Release

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Rust
Approach	Semi-automatic and manual verification

The repository that has been audited includes the **Ref DCL** contract [1].

The auditing process is iterative. Specifically, we will audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following. Our audit report is responsible for the only initial version (`Version 1`), as well as new codes (in the following versions) to fix issues in the audit report.

Project		Commit SHA
Ref DCL Contract	`Version 1`	`0b96a993d6b463ef172f27606c903fe4fc5aaa9c`
	`Version 2`	`876326a1f09bc1ba37cca372196eb3215700d99e`
	`Version 3`	`3e1e1cf814f3ea6321de341dd42200e9bedd19fd`
	`Version 4`	`0c4617f1f1b24348ffd08237f3d9c573dc12fe11`
	`Version 5`	`0564f9926c2aad2892671210a30e3b61f09116bc`
	`Version 6`	`edd130f0a60209b80028b045e460638503af1dc9`
	`Version 7`	`1d301ab8b8e822fb41fdbccb13f0581168555fa9`

Note that, we did **NOT** audit all the modules in the repository. The modules covered by this audit report include **contracts/dcl/src** folder contract only. Specifically, the file covered in this audit include:

- lib.rs
- user_asset.rs
- user.rs
- legacy.rs
- global_config.rs
- utils.rs
- event.rs
- owner.rs
- errors.rs
- api/dcl_liquidity_api.rs
- api/dcl_liquidity_mft.rs
- api/mod.rs
- api/token_receiver.rs
- api/view.rs
- api/dcl_order_api.rs
- api/management.rs
- api/user_asset_api.rs

[1] https://github.com/ref-finance/ref-dcl

- api/nft_approval.rs
- api/nft.rs
- api/storage_api.rs
- api/dcl_api.rs
- dcl/pool.rs
- dcl/dcl_md.rs
- dcl/mod.rs
- dcl/slot_bitmap.rs
- dcl/common_math.rs
- dcl/point_info.rs
- dcl/swap.rs
- dcl/user_mft_asset.rs
- dcl/oracle.rs
- dcl/user_liquidity.rs
- dcl/utils.rs
- dcl/swap_math.rs
- dcl/user_order.rs

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team).

We also manually analyze possible attack scenarios with independent auditors to cross-check the result.

- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

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

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

[2]https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3]https://cwe.mitre.org/

Chapter 2 Findings

In total, we find **fourteen** potential issues. We also have **seventeen** recommendations and **three** notes as follows:

- High Risk: 5
- Medium Risk: 5
- Low Risk: 4
- Recommendations: 17
- Notes: 3

The details are provided in the following sections.

ID	Severity	Description	Category	Status
1	High	Incorrect Storage Check in batch_update_liquidity	Software Security	Fixed
2	Medium	Lack of Check on the withdraw_amount	Software Security	Confirmed
3	High	Non-withdrawable Fees Charged by the Protocol	DeFi Security	Fixed
4	High	Incorrect sqrt_price_loc_96 Calculation in y_swap_x_range_complete_desire()	DeFi Security	Fixed
5	Low	Liquidity on Endpoint Processed Before the Limit Order	DeFi Security	Fixed
6	Medium	Potential Failure in the Callback Function	DeFi Security	Fixed
7	Medium	Improper Rounding Implementation	DeFi Security	Fixed
8	Low	Improper Implementation of internal_mft_transfer()	DeFi Security	Fixed
9	Medium	Lack of Check on Remaining Mft when Updating Farming Contract	DeFi Security	Fixed
10	Medium	Inappropriate Limitation of mft_assets for Farming Contract	DeFi Security	Fixed
11	Low	Lack of Pausable Feature	DeFi Security	Confirmed
12	Low	Liquidity on Endpoint Processed Before the Limit Order	DeFi Security	Fixed
13	High	Lack of Check on Repeated Liquidity in internal_check_remove_liquidity_infos	DeFi Security	Fixed
14	High	Unchecked Received Token in internal_add_order	DeFi Security	Fixed
15	-	Potential Elastic Supply Token Problem	Recommendation	Confirmed
16	-	Potential Centralization Problem	Recommendation	Confirmed
17	-	Redundant Code	Recommendation	Fixed
18	-	Gas Optimization	Recommendation	Fixed
19	-	Unused Code	Recommendation	Fixed
20	-	Repeated Variable Assignments	Recommendation	Fixed
21	-	Incomplete Implementation of Function cancel_order()	Recommendation	Fixed
22	-	Code Optimization	Recommendation	Confirmed
23	-	Unsupported Token Frozen List	Recommendation	Fixed
24	-	Redundant Clone in nft_transfer_call()	Recommendation	Fixed
25	-	Redundant Information in MftId	Recommendation	Confirmed
26	-	Lack of Check on Duplicate Tokens in Frozen List	Recommendation	Confirmed
27	-	Potential Failure of NEAR Transfer	Recommendation	Confirmed
28	-	Skipped Transfer in Function storage_deposit and storage_deposit	Recommendation	Fixed
29	-	Lack of Check on Empty Argument	Recommendation	Fixed
30	-	Spelling Error	Recommendation	Fixed
31	-	Redundant Event Emission in View Functions	Recommendation	Fixed
32	-	Assumption on the Secure Implementation of Contract Dependencies	Notes	Confirmed
33	-	Unsupported Increasement of Selling Tokens for Limit Orders	Notes	Confirmed
34	-	Unsupported Deposit of Native NEAR Tokens	Notes	Confirmed

2.1 Software Security

2.1.1 Incorrect Storage Check in batch_update_liquidity

Severity High

Status Fixed in `Version 5`

Introduced by `Version 4`

Description Function `batch_update_liquidity()` allows the user to add `liquidity` and remove `liquidity` in a batch processing. Since the storage used by the user may be changed in this process, the function verifies that the user's available slot plus the `remove_liquidity_infos.len()` is greater than the `add_liquidity_infos.len()`. However, it is worth noting that removing liquidity may not necessarily lead to an increase in user's available slots. In this case, this check may not be entirely accurate.

```rust
pub fn batch_update_liquidity(
    &mut self,
    remove_liquidity_infos: Vec<RemoveLiquidityInfo>,
    add_liquidity_infos: Vec<AddLiquidityInfo>,
    skip_unwrap_near: Option<bool>
) {
    require!(remove_liquidity_infos.len() > 0 && add_liquidity_infos.len() > 0);
    self.assert_contract_running();
    let user_id = env::predecessor_account_id();
    let mut user = self.internal_unwrap_user(&user_id);
    let global_config = self.internal_get_global_config();
    require!(user.get_available_slots(global_config.storage_price_per_slot, global_config.
        storage_for_asset) + remove_liquidity_infos.len() as u64 >= add_liquidity_infos.len()
        as u64, E107_NOT_ENOUGH_STORAGE_FOR_SLOTS);

    let mut pool_cache = HashMap::new();
    let mut liquiditys = vec![];
    let remove_mft_details = self.internal_check_remove_liquidity_infos(&mut user, &mut
        liquiditys, &mut pool_cache, &remove_liquidity_infos);
    for (mft_id, v_liquidity) in remove_mft_details {
        self.internal_decrease_mft_supply(&mft_id, v_liquidity);
    }
    let refund_tokens = self.internal_batch_remove_liquidity(&user_id, &mut pool_cache, &mut
        liquiditys, remove_liquidity_infos);
    for (token_id, amount) in refund_tokens.into_iter() {
        user.add_asset(&token_id, amount);}

    self.internal_update_or_burn_liquiditys(&mut user, liquiditys);

    let mut lpt_ids = vec![];
    let mut inner_id = self.data_mut().latest_liquidity_id;
    self.internal_check_add_liquidity_infos(&mut user, &mut lpt_ids, &mut pool_cache, &mut
        inner_id, &add_liquidity_infos);
    self.data_mut().latest_liquidity_id = inner_id;


    let (refund_tokens, liquiditys) = self.internal_batch_add_liquidity(&user_id, &lpt_ids, &
        mut pool_cache, add_liquidity_infos);
    
    for (token_id, amount) in refund_tokens {
        self.process_transfer(&user_id, &token_id, amount, skip_unwrap_near);
    }

    for (pool_id, pool) in pool_cache {
        self.internal_set_pool(&pool_id, pool);
    }

    self.internal_mint_liquiditys(user, liquiditys);
}
```

Listing 2.1: contracts/dcl/src/api/dcl_liquidity_api.rs

Impact Users can bypass the limit on the number of added `liquidity`.

Suggestion Check the available slots correctly.

2.1.2 Lack of Check on the withdraw_amount

Severity Medium

Status Confirmed

Introduced by `Version 4`

Description In the function `storage_deposit()`, the minimum amount of `NEAR` to be deposited for new users is set as `STORAGE_BALANCE_MIN_BOUND`. However, users are allowed to withdraw `NEAR`s via the function `storage_withdraw()` from their accounts, even if the remaining balance is less than `STORAGE_BALANCE_MIN_BOUND`.

```
    #[payable]
    fn storage_withdraw(
        &mut self,
        amount: Option<U128>,
    ) -> StorageBalance {
        assert_one_yocto();
        self.assert_contract_running();

        let account_id = env::predecessor_account_id();
        let mut user = self.internal_unwrap_user(&account_id);
        let receiver_id = user.sponsor_id.clone();
        let global_config = self.internal_get_global_config();
        let storage_price_per_slot = global_config.storage_price_per_slot;
        let available_slots = user.get_available_slots(storage_price_per_slot, global_config.
            storage_for_asset);

        let max_amount = available_slots as u128 * storage_price_per_slot;
        let withdraw_amount = if let Some(a) = amount {
            if a.0 > max_amount { max_amount } else { a.0 }
        } else {
            max_amount
        };

        user.locked_near_for_storage -= withdraw_amount;

        Event::WithdrawUserStorage {
            operator: &account_id,
            receiver: &receiver_id,
            amount: &U128(withdraw_amount),
            remain: &U128(user.locked_near_for_storage),
        }.emit();

        self.internal_set_user(&account_id, user);

        if withdraw_amount > 0 {
            Promise::new(receiver_id).transfer(withdraw_amount);
        }

        self.storage_balance_of(account_id).unwrap()
    }
```

Listing 2.2: contracts/dcl/src/api/storage_api.rs

Impact User's storage fee can be less than `STORAGE_BALANCE_MIN_BOUND`.

Suggestion Add a check to ensure the deposited amount of the user will be at least `STORAGE_BALANCE_MIN_BOUND` after the withdrawal.

Feedback from the Project The contract's design enables the removal of all idle slot fees, including some slot fees that were pre-deposited during registration.

2.2 DeFi Security

2.2.1 Non-withdrawable Fees Charged by the Protocol

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description `total_fee_x_charged` and `total_fee_y_charged` (lines 27-30) are used to record the charged protocol fees during the swap actions. However, the protocol fee can not be withdrawn due to the lack of corresponding functions.

```
3   #[derive(BorshSerialize, BorshDeserialize, Serialize)]
4   #[serde(crate = "near_sdk::serde")]
5   pub struct Pool {
6       pub pool_id: PoolId,
7       pub token_x: AccountId,
8       pub token_y: AccountId,
9       pub fee: u32,
10      pub point_delta: i32,
11
12      pub current_point: i32,
13      #[serde(skip_serializing)]
14      pub sqrt_price_96: U256,
15      #[serde(with = "u128_dec_format")]
```

```
        pub liquidity: u128,
        #[serde(with = "u128_dec_format")]
        pub liquidity_x: u128,
        #[serde(with = "u128_dec_format")]
        pub max_liquidity_per_point: u128,

        #[serde(skip_serializing)]
        pub fee_scale_x_128: U256, // token X fee per unit of liquidity
        #[serde(skip_serializing)]
        pub fee_scale_y_128: U256, // token Y fee per unit of liquidity

        #[serde(skip_serializing)]
        pub total_fee_x_charged: U256,
        #[serde(skip_serializing)]
        pub total_fee_y_charged: U256,

        #[serde(with = "u256_dec_format")]
        pub volume_x_in: U256,
        #[serde(with = "u256_dec_format")]
        pub volume_y_in: U256,
        #[serde(with = "u256_dec_format")]
        pub volume_x_out: U256,
        #[serde(with = "u256_dec_format")]
        pub volume_y_out: U256,

        #[serde(with = "u128_dec_format")]
        pub total_liquidity: u128,
        #[serde(with = "u128_dec_format")]
        pub total_order_x: u128,
        #[serde(with = "u128_dec_format")]
        pub total_order_y: u128,
        #[serde(with = "u128_dec_format")]
        pub total_x: u128,
        #[serde(with = "u128_dec_format")]
        pub total_y: u128,

        #[serde(skip_serializing)]
        pub point_info: PointInfo,
        #[serde(skip_serializing)]
        pub slot_bitmap: SlotBitmap,

        pub state: RunningState,
    }
```

Listing 2.3: contracts/dcl/src/pool.rs

Impact Protocol fees are locked in the contract.

Suggestion Implement the corresponding withdrawal functions.

2.2.2 Incorrect sqrt_price_loc_96 Calculation in y_swap_x_range_complete_desire()

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In function `y_swap_x_range_complete_desire()`, the calculation of `sqrt_price_loc_96` is wrong. According to the current implementation, the `result.loc_pt` calculated from the `sqrt_price_loc_96` is the offset relative to the `left_point`. However, the correct `result.loc_pt` should be the offset relative to the point zero. In this case, the numerator in line 692 should be `sqrt_price_r_96` instead of `sqrt_price_pr_pl_96`.

```
662 /// try to swap from right to left in range [left_point, right_point) with all liquidity used.
663 /// @param liquidity: liquidity of each point in the range
664 /// @param sqrt_price_l_96: sqrt of left point price in 2^96 power
665 /// @param left_point: left point of this range
666 /// @param sqrt_price_r_96: sqrt of right point price in 2^96 power
667 /// @param right_point: right point of this range
668 /// @param desire_x: amount of token X as swap-out
669 /// @return Y2XRangeCompRetDesire
670 pub fn y_swap_x_range_complete_desire(
671     liquidity: u128,
672     sqrt_price_l_96: U256,
673     left_point: i32,
674     sqrt_price_r_96: U256,
675     right_point: i32,
676     desire_x: u128
677 ) -> Y2XRangeCompRetDesire {
678     let mut result = Y2XRangeCompRetDesire::default();
679     let max_x = get_amount_x(liquidity, left_point, right_point, sqrt_price_r_96, sqrt_rate_96(),
          false).as_u128();
680     if max_x <= desire_x {
681         // maxX <= desireX <= uint128.max
682         result.acquire_x = max_x;
683         result.cost_y = get_amount_y(liquidity, sqrt_price_l_96, sqrt_price_r_96, sqrt_rate_96(),
              true);
684         result.complete_liquidity = true;
685         return result;
686     }
687
688     let sqrt_price_pr_pl_96 = get_sqrt_price(right_point - left_point);
689     let sqrt_price_pr_m1_96 = sqrt_price_r_96.mul_fraction_floor(pow_96(), sqrt_rate_96());
690     let div = sqrt_price_pr_pl_96 - U256::from(desire_x).mul_fraction_floor(sqrt_price_r_96 -
          sqrt_price_pr_m1_96, U256::from(liquidity));
691
692     let sqrt_price_loc_96 = sqrt_price_pr_pl_96.mul_fraction_floor(pow_96(), div);
693
694     result.complete_liquidity = false;
695     result.loc_pt = get_log_sqrt_price_floor(sqrt_price_loc_96);
696
697     result.loc_pt = std::cmp::max(left_point, result.loc_pt);
698     result.loc_pt = std::cmp::min(right_point - 1, result.loc_pt);
699     result.sqrt_loc_96 = get_sqrt_price(result.loc_pt);
700
701     if result.loc_pt == left_point {
702         result.acquire_x = 0;
```

```
        result.cost_y = Default::default();
        return result;
    }
    result.complete_liquidity = false;
    result.acquire_x = std::cmp::min(
        get_amount_x(liquidity, left_point, result.loc_pt, result.sqrt_loc_96, sqrt_rate_96(),
            false).as_u128(),
        desire_x);

    result.cost_y = get_amount_y(liquidity, sqrt_price_l_96, result.sqrt_loc_96, sqrt_rate_96(),
        true);
    result
}
```

Listing 2.4: contracts/dcl/src/swap_math.rs

For example, we have a liquidity whose range is from the `left_point (A)` to the `result.loc_pt (B)`, `L` denotes the amount of liquidity and `X` denotes the desired amount for token `X`.

Now we have:

$$\frac{L}{\sqrt{1.0001}^{A}} + \frac{L}{\sqrt{1.0001}^{A+1}} + \frac{L}{\sqrt{1.0001}^{A+2}} \cdots + \frac{L}{\sqrt{1.0001}^{B-1}} = X$$

With D =1.0001, the formula (a) can be simplified as follows :

$$L * \frac{1 - D^{A-B}}{D^{A} - D^{A-1}} = X$$

$$L * D^{A-B} = L - X(D^{A} - D^{A-1})$$

$$D^{B-A} = \frac{L}{L - X(D^{A} - D^{A-1})}$$

For `result.loc_pt,` we have:

$$B = log_{D}\frac{L}{L - X(D^{A} - D^{A-1})} + A$$

However, the current implementation of Ref-DCL for calculating `result.loc_pt` is:

$$B = log_{D}\frac{D^{C-A}}{D^{C-A} - \frac{X}{L} * (D^{C} - D^{C-1})}$$

where C denotes the `right_point`

$$B = log_{D}\frac{L * D^{C-A}}{L * D^{C-A} - X * (D^{C} - D^{C-1})}$$

$$B = log_{D}\frac{L}{L - X * (D^{A} - D^{A-1})}$$

The `result.loc_pt` calculated from `Ref-DCL` is incorrect, and the correct calculation should follow the equation (e).

Impact There won't be enough `token_x` swapped out due to the incorrect calculation described above.

Suggestion Replace the `sqrt_price_pr_pl_96` with the `sqrt_price_r_96` when calculating the `sqrt_price_loc_96` in function `y_swap_x_range_complete_desired()`.

2.2.3 Liquidity on Endpoint Processed Before the Limit Order

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Function `internal_x_swap_y()` is to swap `token_x` to `token_y`. During the swapping process, the liquidity will be processed before the limit order. In this case, when the point stops at the `next_point`, which is an `endpoint`, and the amount of `token_x` is not fully swapped, the liquidity can be used up while the order is not processed. This is inconsistent with the original design.

```rust
/// Process x_swap_y in range
/// @param protocol_fee_rate
/// @param input_amount: amount of token X
/// @param low_boundary_point
/// @param is_quote: whether the quote function is calling
/// @return (consumed_x, gained_y, is_finished)
pub fn internal_x_swap_y(&mut self, protocol_fee_rate: u32, input_amount: u128,
    low_boundary_point: i32, is_quote: bool) -> (u128, u128, bool) {
    let boundary_point = std::cmp::max(low_boundary_point, LEFT_MOST_POINT);
    let mut amount = input_amount;
    let mut amount_x = 0;
    let mut amount_y = 0;
    let mut is_finished = false;
    let mut current_order_or_endpt = self.point_info.get_point_type_value(self.current_point,
        self.point_delta);

    while boundary_point <= self.current_point && !is_finished {
        if current_order_or_endpt & 2 > 0 {
            // process limit order
            let mut point_data = self.point_info.0.get(&self.current_point).unwrap();
            let mut order_data = point_data.order_data.take().unwrap();
            let process_ret = self.process_limit_order_y(protocol_fee_rate, &mut order_data,
                amount);
            is_finished = process_ret.0;
            (amount, amount_x, amount_y) = (amount-process_ret.1, amount_x+process_ret.1,
                amount_y+process_ret.2);

            self.update_order(&mut point_data, order_data, is_quote);

            if is_finished {
                break;
            }
        }

        let search_start = self.current_point - 1;

        if current_order_or_endpt & 1 > 0 {
            // current point is an liquidity endpoint, process liquidity
```

```
                let process_ret = self.process_liquidity_y(protocol_fee_rate, amount, self.
                    current_point);
                is_finished = process_ret.0;
                (amount, amount_x, amount_y) = (amount-process_ret.1, amount_x+process_ret.1,
                    amount_y+process_ret.2);

                if !is_finished {
                    // pass endpoint
                    self.pass_endpoint(self.current_point, is_quote, true);
                    // move one step to the left
                    self.current_point -= 1;
                    self.sqrt_price_96 = get_sqrt_price(self.current_point);
                    self.liquidity_x = 0;
                }
                if is_finished || self.current_point < boundary_point {
                    break;
                }
            }

            // process range liquidity
            let next_pt= match self.slot_bitmap.get_nearest_left_valued_slot(search_start, self.
                point_delta, boundary_point / self.point_delta){
                Some(point) => {
                    if point < boundary_point {
                        boundary_point
                    } else {
                        point
                    }
                 },
                None => { boundary_point }
            };

            let process_ret = self.process_liquidity_y(protocol_fee_rate, amount, next_pt);
            is_finished = process_ret.0;
            (amount, amount_x, amount_y) = (amount-process_ret.1, amount_x+process_ret.1, amount_y+
                process_ret.2);

            if self.current_point == next_pt {
                current_order_or_endpt = self.point_info.get_point_type_value(next_pt, self.
                    point_delta);
            } else {
                current_order_or_endpt = 0;
            }


            if self.current_point <= boundary_point {
                if self.current_point == boundary_point && !is_finished && current_order_or_endpt &
                     2 > 0 {
                    // this final point should check if there is limit order to trade
                    let mut point_data = self.point_info.0.get(&self.current_point).unwrap();
                    let mut order_data = point_data.order_data.take().unwrap();
                    let process_ret = self.process_limit_order_y(protocol_fee_rate, &mut order_data,
                        amount);
                    is_finished = process_ret.0;
                    (_, amount_x, amount_y) = (amount-process_ret.1, amount_x+process_ret.1, amount_y+process_ret.2);

                    if !is_quote {
                        point_data.order_data = Some(order_data);
                        self.point_info.0.insert(&self.current_point, &point_data);
                        if order_data.selling_x == 0 && order_data.selling_y == 0 && current_order_or_endpt & 1 == 0 {
                            self.slot_bitmap.set_zero(self.current_point, self.point_delta);
                        }
                    }
                }
                break;
            }
        }
        (amount_x, amount_y, is_finished)
    }
```

Listing 2.5: contracts/dcl/src/pool.rs

Impact Liquidity on the endpoint may be swapped out before the limit order on the same `endpoint`.

Suggestion Process the `liquidity_y` that ranges from the `current_point` to the `next_point+1` first, if there're still some `token_x` left, move to the `next_point`, and handle the limit order before the liquidity on the point.

2.2.4 Potential Failure in the Callback Function

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In function `callback_post_withdraw_asset()`, if the `PosmiseResult` is checked as `Failed` and the number of the user's assets has reached the threshold, this callback function will panic in line 5 of function `add_asset()`. In this case, the `Event::Lostfound` will not be emitted.

```
    #[private]
    pub fn callback_post_withdraw_asset(
        &mut self,
        token_id: AccountId,
        user_id: AccountId,
        amount: U128,
    ) -> bool {
        require!(
            env::promise_results_count() == 1,
            E001_PROMISE_RESULT_COUNT_INVALID
        );
        let amount: Balance = amount.into();
        match env::promise_result(0) {
            PromiseResult::NotReady => unreachable!(),
            PromiseResult::Successful(_) => {
                true
            }
            PromiseResult::Failed => {
                // This reverts the changes from withdraw function.
                if let Some(mut user) = self.internal_get_user(&user_id) {
                    user.add_asset(&token_id, amount);
                    self.internal_set_user(&user_id, user);

                    Event::Lostfound {
                        user: &user_id,
                        token: &token_id,
                        amount: &U128(amount),
                        locked: &false,
                    }
                    .emit();
                } else {
                    Event::Lostfound {
                        user: &user_id,
                        token: &token_id,
                        amount: &U128(amount),
                        locked: &true,
                    }
                    .emit();
                }
                false
            }
        }
    }
```

Listing 2.6: contracts/dcl/src/user_asset.rs

```
pub fn add_asset(&mut self, token_id: &AccountId, amount: Balance) {
    require!(self.assets.len() < DEFAULT_MAX_USER_ASSET_COUNT || self.assets.get(token_id).is_some
        (), "ERR_USER_ASSET_COUNT_EXCEEDED");
    self.assets.insert(
        token_id,
        &(amount + self.assets.get(token_id).unwrap_or(0_u128)).clone(),
    );
}
```

Listing 2.7: contracts/dcl/src/user_asset.rs

The same problem exists in the function `callback_post_withdraw_near()`.

```
#[private]
pub fn callback_post_withdraw_near(
    &mut self,
    user_id: AccountId,
    amount: U128,
) -> bool {
    require!(
        env::promise_results_count() == 1,
        E001_PROMISE_RESULT_COUNT_INVALID
    );
    let amount: Balance = amount.into();
    match env::promise_result(0) {
        PromiseResult::NotReady => unreachable!(),
        PromiseResult::Successful(_) => {
            Promise::new(user_id).transfer(amount);
            true
        }
        PromiseResult::Failed => {
            // This reverts the changes from withdraw function.
            if let Some(mut user) = self.internal_get_user(&user_id) {
                user.add_asset(&self.data().wnear_id, amount);
                self.internal_set_user(&user_id, user);

                Event::Lostfound {
                    user: &user_id,
                    token: &self.data().wnear_id,
                    amount: &U128(amount),
                    locked: &false,
                }
                .emit();
            } else {
                Event::Lostfound {
                    user: &user_id,
                    token: &self.data().wnear_id,
                    amount: &U128(amount),
                    locked: &true,
                }
                .emit();
            }
            false
        }
    }
}
```

Listing 2.8: contracts/dcl/src/user_asset.rs

Impact Users' assets may be lost due to the potential failure of the callback function.

Suggestion If the function `add_asset()` is called by the callback function and the number of the user's assets has reached the threshold (i.e., 64), emit an `Event::Lostfound` instead of throwing into a panic.

2.2.5 Improper Rounding Implementation

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In function `interrnal_update_order()`, the amount of the `token_x` or `token_y` earned by the user in lines 304-313 is rounded up with function `mul_fraction_ceil()`, which is inconsistent with the calculation in lines 349-358.

```
279     /// Sync user order with point order, try to claim as much earned as possible
280     /// @param ue: user order
```

```
/// @param po: point order
/// @return earned amount this time
pub fn internal_update_order(ue: &mut UserOrder, po: &mut OrderData) -> u128 {
    let is_earn_y = ue.is_earn_y();
    let sqrt_price_96 = get_sqrt_price(ue.point);
    let (total_earn, total_legacy_earn, acc_legacy_earn, cur_acc_earn) = if is_earn_y {
        (
            po.earn_y,
            po.earn_y_legacy,
            po.acc_earn_y_legacy,
            po.acc_earn_y,
        )
    } else {
        (
            po.earn_x,
            po.earn_x_legacy,
            po.acc_earn_x_legacy,
            po.acc_earn_x,
        )
    };

    if ue.last_acc_earn < acc_legacy_earn {
        // this order has been fully filled
        let mut earn = if is_earn_y {
            let liquidity =
                U256::from(ue.remain_amount).mul_fraction_ceil(sqrt_price_96, pow_96());
            liquidity.mul_fraction_ceil(sqrt_price_96, pow_96())
        } else {
            let liquidity =
                U256::from(ue.remain_amount).mul_fraction_ceil(pow_96(), sqrt_price_96);
            liquidity.mul_fraction_ceil(pow_96(), sqrt_price_96)
        }
        .as_u128();

        // update po
        if earn > total_legacy_earn {
            // just protect from some rounding errors
            earn = total_legacy_earn;
        }
        if is_earn_y {
            po.earn_y_legacy -= earn;
        } else {
            po.earn_x_legacy -= earn;
        }

        // update ue
        ue.last_acc_earn = cur_acc_earn;
        ue.remain_amount = 0;
        ue.bought_amount += earn;
        ue.unclaimed_amount = Some(U128(earn));

        earn
    } else {
```

```
334             // this order needs to compete earn
335             let mut earn = min((cur_acc_earn - ue.last_acc_earn).as_u128(), total_earn);
336 
337             let mut sold = if is_earn_y {
338                 let liquidity = U256::from(earn).mul_fraction_ceil(pow_96(), sqrt_price_96);
339                 liquidity.mul_fraction_ceil(pow_96(), sqrt_price_96)
340             } else {
341                 let liquidity = U256::from(earn).mul_fraction_ceil(sqrt_price_96, pow_96());
342                 liquidity.mul_fraction_ceil(sqrt_price_96, pow_96())
343             }
344             .as_u128();
345 
346             // actual sold should less or equal to remaining, adjust sold and earn if needed
347             if sold > ue.remain_amount {
348                 sold = ue.remain_amount;
349                 earn = if is_earn_y {
350                     let liquidity =
351                         U256::from(sold).mul_fraction_floor(sqrt_price_96, pow_96());
352                     liquidity.mul_fraction_floor(sqrt_price_96, pow_96())
353                 } else {
354                     let liquidity =
355                         U256::from(sold).mul_fraction_floor(pow_96(), sqrt_price_96);
356                     liquidity.mul_fraction_floor(pow_96(), sqrt_price_96)
357                 }
358                 .as_u128();
359             }
360 
361             // update po
362             if earn > total_earn {
363                 // just protect from some rounding errors
364                 earn = total_earn;
365             }
366             if is_earn_y {
367                 po.earn_y -= earn;
368             } else {
369                 po.earn_x -= earn;
370             }
371 
372             // update ue
373             ue.last_acc_earn = cur_acc_earn;
374             ue.remain_amount -= sold;
375             ue.bought_amount += earn;
376             ue.unclaimed_amount = Some(U128(earn));
377 
378             earn
379         }
380     }
```

Listing 2.9: contracts/dcl/src/user_order.rs

Impact Some users may earn more tokens while others can not withdraw all the tokens.

Suggestion Use function `mul_fraction_floor()` instead of `mul_fraction_ceil()` when calculating the users' earned tokens in lines 304-313.

2.2.6 Improper Implementation of internal_mft_transfer()

Severity Low

Status Fixed in Version 3

Introduced by Version 2

Description The internal function `internal_mft_transfer()` is implemented to transfer the `mft` tokens between `sender` and `receiver`. However, it does not consider the situation that the `sender` and `receiver` can be the same account. In this case, the `sender`/`receiver` (e.g., farming contract) can mint infinite `mft` tokens by setting `receiver` as the `sender`.

```
fn internal_mft_transfer(
    &mut self,
    token_id: String,
    sender_id: &AccountId,
    receiver_id: &AccountId,
    amount: u128,
    memo: Option<String>,
) {
    let mut sender = self.internal_unwrap_user(sender_id);
    let mut receiver = self.internal_unwrap_user(receiver_id);

    sender.sub_mft_asset(&token_id, amount);
    receiver.add_mft_asset(&token_id, amount);

    self.internal_set_user(sender_id, sender);
    self.internal_set_user(receiver_id, receiver);

    if let Some(memo) = memo {
        log!("Memo: {}", memo);
    }
}
```

Listing 2.10: contracts/dcl/src/multi_fungible_token.rs

Impact Although it can only be done by the `farming` contract, the implementation of the above `internal_mft_transfer` is improper.

Suggestion Add the check to ensure the `sender` and the `receiver` are not the same account.

2.2.7 Lack of Check on Remaining Mft when Updating Farming Contract

Severity Medium

Status Fixed in Version 3

Introduced by Version 2

Description The function `set_farming_contract_id()` is used to set and update the `farming` contract. However, there is no check on whether there still exists some `mft` tokens in the previous `farming` contract. In this case, the `mft` tokens can be locked in the previous `farming` contract.

```
#[payable]
    pub fn set_farming_contract_id(&mut self, farming_contract_id: AccountId) {
        assert_one_yocto();
        self.assert_owner();

        if !self.data().users.contains_key(&farming_contract_id) {
            self.data_mut().users.insert(
                &farming_contract_id,
                &User::new(&farming_contract_id, &env::current_account_id()).into(),
            );
            self.data_mut().user_count += 1;
        }

        self.data_mut().farming_contract_id = farming_contract_id;
    }
```

Listing 2.11: contracts/dcl/src/owner.rs

Impact Users' `mft` can be locked in the previous `farming` contract.

Suggestion Add the check to ensure no `mft` tokens left in the previous `farming` contract before updating.

2.2.8 Inappropriate Limitation of mft_assets for Farming Contract

Severity Medium

Status Fixed in `Version 3`

Introduced by `Version 2`

Description There is a check in the function `add_mft_asset()` to ensure the amount of the user's `mft` assets will be no larger than `DEFAULT_MAX_USER_ASSET_COUNT` (i.e. 64). However, the `farming` contract, which should hold much more assets than the normal user, is also limited by this number, which is inappropriate.

```
    pub fn add_mft_asset(&mut self, mft_id: &MftId, amount: Balance) {
        require!(self.mft_assets.len() < DEFAULT_MAX_USER_ASSET_COUNT || self.mft_assets.get(mft_id
            ).is_some(), "ERR_USER_ASSET_COUNT_EXCEEDED");
        self.add_mft_asset_uncheck(mft_id, amount);
    }
```

Listing 2.12: contracts/dcl/src/user_mft_asset.rs

Impact The `farming` contract will not be able to receive `mft` tokens after the amount of the `mft` assets reaches the cap.

Suggestion The amount of the `mft` tokens for the `farming` contract should be limited by a different value.

2.2.9 Lack of Pausable Feature

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In current implementation, even if one of the pools is paused, the user can still add liquidity, append liquidity, merge liquidity, remove liquidity, add order, and cancel order to the paused pool.

```
pub fn internal_add_order(
    &mut self,
    client_id: String,
    user_id: &AccountId,
    token_id: &AccountId,
    amount: Balance,
    pool_id: &PoolId,
    point: i32,
    buy_token: &AccountId,
    swapped_amount: Balance,
    swap_earn_amount: Balance,
) -> OrderId {
    let mut pool = self.internal_get_pool(pool_id).unwrap();
    self.assert_no_frozen_tokens(&[pool.token_x.clone(), pool.token_y.clone()]);
    require!(point % pool.point_delta as i32 == 0, E202_ILLEGAL_POINT);
    require!(client_id.len() <= MAX_USER_ORDER_CLIENT_ID_LEN, E306_INVALID_CLIENT_ID);
    require!(amount - swapped_amount > 0, E307_INVALID_SELLING_AMOUNT);

    let mut user = self.internal_unwrap_user(user_id);
    let order_key = gen_user_order_key(pool_id, point);
    require!(
        user.order_keys.get(&order_key).is_none(),
        E301_ACTIVE_ORDER_ALREADY_EXIST
    );
    require!(
        user.order_keys.len() < DEFAULT_MAX_USER_ACTIVE_ORDER_COUNT,
        E302_USER_ACTIVE_ORDER_NUM_EXCEEDED
    );

    let mut point_data = pool.point_info.0.get(&point).unwrap_or_default();
    let prev_active_order = point_data.has_active_order();
    let mut point_order: OrderData = point_data.order_data.unwrap_or_default();

    let order_id = gen_order_id(pool_id, &mut self.data_mut().latest_order_id);
    let mut order = UserOrder {
        client_id,
        order_id: order_id.clone(),
        owner_id: user_id.clone(),
        pool_id: pool_id.clone(),
        point,
        sell_token: token_id.clone(),
        buy_token: buy_token.clone(),
        original_deposit_amount: amount,
        swap_earn_amount,
        original_amount: amount - swapped_amount,
        created_at: env::block_timestamp(),
        last_acc_earn: U256::zero(),
        remain_amount: amount - swapped_amount,
        cancel_amount: 0_u128,
        bought_amount: 0_u128,
        unclaimed_amount: None,
    };
```

```

        let (token_x, token_y, _) = pool_id.parse_pool_id();
        if token_x == (*token_id) {
            require!(buy_token == &token_y, E303_ILLEGAL_BUY_TOKEN);
            require!(point >= pool.current_point, E202_ILLEGAL_POINT); // greater or equal to
                current point
            require!(point <= RIGHT_MOST_POINT, E202_ILLEGAL_POINT);
            order.last_acc_earn = point_order.acc_earn_y;
            point_order.selling_x += amount - swapped_amount;
            pool.total_x += amount - swapped_amount;
            pool.total_order_x += amount - swapped_amount;
        } else {
            require!(buy_token == &token_x, E303_ILLEGAL_BUY_TOKEN);
            require!(point <= pool.current_point, E202_ILLEGAL_POINT); // less or equal to current
                point
            require!(point >= LEFT_MOST_POINT, E202_ILLEGAL_POINT);
            order.last_acc_earn = point_order.acc_earn_x;
            point_order.selling_y += amount - swapped_amount;
            pool.total_y += amount - swapped_amount;
            pool.total_order_y += amount - swapped_amount;
        }
        point_order.user_order_count += 1;
        // update order
        user.order_keys.insert(&order_key, &order.order_id);
        self.internal_set_user(user_id, user);

        // update pool info
        point_data.order_data = Some(point_order);
        pool.point_info.0.insert(&point, &point_data);
        if !prev_active_order && !point_data.has_active_liquidity() {
            pool.slot_bitmap.set_one(point, pool.point_delta);
        }
        self.internal_set_pool(pool_id, pool);

        Event::OrderAdded {
            order_id: &order.order_id,
            created_at: &U64(env::block_timestamp()),
            owner_id: &order.owner_id,
            pool_id: &order.pool_id,
            point: &order.point,
            sell_token: &order.sell_token,
            buy_token: &order.buy_token,
            original_amount: &U128(order.original_amount),
            original_deposit_amount: &U128(order.original_deposit_amount),
            swap_earn_amount: &U128(order.swap_earn_amount),
        }
        .emit();
        self.internal_set_user_order(&order_id, order);

        order_id
    }
```

Listing 2.13: contracts/dcl/src/user_order.rs

```rust
pub fn cancel_order(&mut self, order_id: OrderId, amount: Option<U128>) -> (U128, U128) {
        self.assert_contract_running();
        let mut order = self.internal_unwrap_user_order(&order_id);

        let user_id = env::predecessor_account_id();
        require!(order.owner_id == user_id, E300_NOT_ORDER_OWNER);

        let mut pool = self.internal_get_pool(&order.pool_id).unwrap();
        self.assert_no_frozen_tokens(&[pool.token_x.clone(), pool.token_y.clone()]);
        let mut point_data = pool.point_info.0.get(&order.point).unwrap();
        let mut point_order: OrderData = point_data.order_data.unwrap();

        let earned = internal_update_order(&mut order, &mut point_order);

        // do cancel
        let actual_cancel_amount = if let Some(expected_cancel_amount) = amount {
            min(expected_cancel_amount.into(), order.remain_amount)
        } else {
            order.remain_amount
        };
        order.cancel_amount += actual_cancel_amount;
        order.remain_amount -= actual_cancel_amount;

        // update point_data
        if order.is_earn_y() {
            pool.total_x -= actual_cancel_amount;
            pool.total_y -= earned;
            pool.total_order_x -= actual_cancel_amount;
            point_order.selling_x -= actual_cancel_amount;
        } else {
            pool.total_x -= earned;
            pool.total_y -= actual_cancel_amount;
            pool.total_order_y -= actual_cancel_amount;
            point_order.selling_y -= actual_cancel_amount;
        }
        point_data.order_data = if order.remain_amount == 0 {
            point_order.user_order_count -= 1;
            if point_order.user_order_count == 0 {
                pool.total_order_x -= point_order.selling_x;
                pool.total_order_y -= point_order.selling_y;
                pool.total_x -= point_order.selling_x;
                pool.total_y -= point_order.selling_y;
                None
            } else {
                Some(point_order)
            }
        } else {
            Some(point_order)
        };
        if !point_data.has_active_liquidity() && !point_data.has_active_order() {
            pool.slot_bitmap.set_zero(order.point, pool.point_delta);
        }
        if point_data.has_order() || point_data.has_liquidity() {
            pool.point_info.0.insert(&order.point, &point_data);
        } else {
            pool.point_info.0.remove(&order.point);
        }
        self.internal_set_pool(&order.pool_id, pool);

        Event::OrderCancelled {
            order_id: &order.order_id,
            created_at: &U64(order.created_at),
            cancel_at: &U64(env::block_timestamp()),
            owner_id: &order.owner_id,
            pool_id: &order.pool_id,
            point: &order.point,
            sell_token: &order.sell_token,
            buy_token: &order.buy_token,
            request_cancel_amount: &amount,
            actual_cancel_amount: &U128(actual_cancel_amount),
            original_amount: &U128(order.original_amount),
            cancel_amount: &U128(order.cancel_amount),
            remain_amount: &U128(order.remain_amount),
            bought_amount: &U128(order.bought_amount),
        }
        .emit();

        // transfer token to user
        if earned > 0 {
            if order.buy_token == self.data().wnear_id {
                self.process_near_transfer(&order.owner_id, earned);
            } else {
                self.process_ft_transfer(&order.owner_id, &order.buy_token, earned);
            }
        }

        if actual_cancel_amount > 0 {
            if order.sell_token == self.data().wnear_id {
                self.process_near_transfer(&order.owner_id, actual_cancel_amount);
            } else {
                self.process_ft_transfer(&order.owner_id, &order.sell_token, actual_cancel_amount);
            }
        }

        // deactive order if needed
        if order.remain_amount == 0 {
            // completed order move to user history
            let order_key = gen_user_order_key(&order.pool_id, order.point);
            let mut user = self.internal_unwrap_user(&user_id);
            user.order_keys.remove(&order_key);
            if user.completed_order_count < DEFAULT_USER_ORDER_HISTORY_LEN {
                user.history_orders.push(&order);
            } else {
                let index = user.completed_order_count % DEFAULT_USER_ORDER_HISTORY_LEN;
                user.history_orders.replace(index, &order);
            }
            user.completed_order_count += 1;
            self.internal_set_user(&user_id, user);
            self.data_mut().user_orders.remove(&order_id);
            Event::OrderCompleted {
                order_id: &order.order_id,
                created_at: &U64(order.created_at),
                completed_at: &U64(env::block_timestamp()),
                owner_id: &order.owner_id,
                pool_id: &order.pool_id,
                point: &order.point,
                sell_token: &order.sell_token,
                buy_token: &order.buy_token,
                original_amount: &U128(order.original_amount),
                original_deposit_amount: &U128(order.original_deposit_amount),
                swap_earn_amount: &U128(order.swap_earn_amount),
                cancel_amount: &U128(order.cancel_amount),
                bought_amount: &U128(order.bought_amount),
            }
            .emit();
        } else {
            self.internal_set_user_order(&order_id, order);
        }

        (actual_cancel_amount.into(), earned.into())
    }
```

Listing 2.14: contracts/dcl/src/user_order.rs

```
pub fn add_liquidity(
        &mut self,
        pool_id: PoolId,
        left_point: i32,
        right_point: i32,
        amount_x: U128,
        amount_y: U128,
        min_amount_x: U128,
        min_amount_y: U128,
    ) -> LptId {
        self.assert_contract_running();
        let user_id = env::predecessor_account_id();
        let mut user = self.internal_unwrap_user(&user_id);
        require!(
            user.liquidity_keys.len() < DEFAULT_MAX_USER_LIQUIDITY_COUNT,
            E217_USER_LIQUIDITY_COUNT_EXCEEDED
        );

        let mut pool = self.internal_unwrap_pool(&pool_id);
        self.assert_no_frozen_tokens(&[pool.token_x.clone(), pool.token_y.clone()]);
        require!(left_point % pool.point_delta == 0 && right_point % pool.point_delta == 0,
             E200_INVALID_ENDPOINT);
        require!(right_point > left_point, E202_ILLEGAL_POINT);
        require!(right_point - left_point < RIGHT_MOST_POINT, E202_ILLEGAL_POINT);
        require!(left_point >= LEFT_MOST_POINT && right_point <= RIGHT_MOST_POINT,
            E202_ILLEGAL_POINT);

        let (new_liquidity, need_x, need_y, acc_fee_x_in_128, acc_fee_y_in_128) = pool.
            internal_add_liquidity(left_point, right_point, amount_x.0, amount_y.0, min_amount_x
            .0, min_amount_y.0);
        user.sub_asset(&pool.token_x, amount_x.0);
        user.sub_asset(&pool.token_y, amount_y.0);

        let lpt_id = gen_lpt_id(&pool_id, &mut self.data_mut().latest_liquidity_id);
        let liquidity = UserLiquidity {
            lpt_id: lpt_id.clone(),
            owner_id: user_id.clone(),
            pool_id: pool_id.clone(),
            left_point,
            right_point,
            last_fee_scale_x_128: acc_fee_x_in_128,
            last_fee_scale_y_128: acc_fee_y_in_128,
            amount: new_liquidity,
            mft_id: String::new(),
            v_liquidity: 0,
            unclaimed_fee_x: None,
            unclaimed_fee_y: None,
        };

        pool.total_liquidity += new_liquidity;
        pool.total_x += need_x;
        pool.total_y += need_y;

        let refund_x = amount_x.0 - need_x;
        let refund_y = amount_y.0 - need_y;
        if refund_x > 0{
            if pool.token_x == self.data().wnear_id {
                self.process_near_transfer(&user_id, refund_x);
            } else {
                self.process_ft_transfer(&user_id, &pool.token_x, refund_x);
            }
        }
        if refund_y > 0{
            if pool.token_y == self.data().wnear_id {
                self.process_near_transfer(&user_id, refund_y);
            } else {
                self.process_ft_transfer(&user_id, &pool.token_y, refund_y);
            }
        }

        self.internal_set_pool(&pool_id, pool);
        Event::LiquidityAdded {
            lpt_id: &lpt_id,
            owner_id: &user_id,
            pool_id: &pool_id,
            left_point: &left_point,
            right_point: &right_point,
            added_amount: &U128(new_liquidity),
            cur_amount: &U128(liquidity.amount),
            paid_token_x: &U128(need_x),
            paid_token_y: &U128(need_y),
        }
        .emit();
        self.internal_mint_liquidity(user, liquidity);
        lpt_id
    }
```

Listing 2.15: contracts/dcl/src/user_liquidity.rs

```
pub fn append_liquidity(
        &mut self,
        lpt_id: LptId,
        amount_x: U128,
        amount_y: U128,
        min_amount_x: U128,
        min_amount_y: U128,
    ) {
        self.assert_contract_running();
        let user_id = env::predecessor_account_id();
        let mut user = self.internal_unwrap_user(&user_id);
        let mut liquidity = self.internal_unwrap_user_liquidity(&lpt_id);
        require!(!liquidity.is_mining(), E218_USER_LIQUIDITY_IS_MINING);
        require!(user_id == liquidity.owner_id, E215_NOT_LIQUIDITY_OWNER);
        let mut pool = self.internal_unwrap_pool(&liquidity.pool_id);
        self.assert_no_frozen_tokens(&[pool.token_x.clone(), pool.token_y.clone()]);

        let (new_liquidity, need_x, need_y, acc_fee_x_in_128, acc_fee_y_in_128) = pool.
            internal_add_liquidity(liquidity.left_point, liquidity.right_point, amount_x.0,
            amount_y.0, min_amount_x.0, min_amount_y.0);
        user.sub_asset(&pool.token_x, amount_x.0);
        user.sub_asset(&pool.token_y, amount_y.0);

        liquidity.get_unclaimed_fee(acc_fee_x_in_128, acc_fee_y_in_128);
        let new_fee_x = liquidity.unclaimed_fee_x.unwrap_or(U128(0)).0;
        let new_fee_y = liquidity.unclaimed_fee_y.unwrap_or(U128(0)).0;

        pool.total_liquidity += new_liquidity;
        pool.total_x += need_x;
        pool.total_y += need_y;
        pool.total_x -= new_fee_x;
        pool.total_y -= new_fee_y;

        // refund
        let refund_x = amount_x.0 - need_x + new_fee_x;
        let refund_y = amount_y.0 - need_y + new_fee_y;
        if refund_x > 0{
            if pool.token_x == self.data().wnear_id {
                self.process_near_transfer(&user_id, refund_x);
            } else {
                self.process_ft_transfer(&user_id, &pool.token_x, refund_x);
            }
        }
        if refund_y > 0{
            if pool.token_y == self.data().wnear_id {
                self.process_near_transfer(&user_id, refund_y);
            } else {
                self.process_ft_transfer(&user_id, &pool.token_y, refund_y);
            }
        }
        // update lpt
        liquidity.amount += new_liquidity;
        liquidity.last_fee_scale_x_128 = acc_fee_x_in_128;
        liquidity.last_fee_scale_y_128 = acc_fee_y_in_128;
        self.internal_set_user(&user.user_id.clone(), user);
        self.internal_set_pool(&liquidity.pool_id, pool);
        Event::LiquidityAppend {
            lpt_id: &lpt_id,
            owner_id: &user_id,
            pool_id: &liquidity.pool_id,
            left_point: &liquidity.left_point,
            right_point: &liquidity.right_point,
            added_amount: &U128(new_liquidity),
            cur_amount: &U128(liquidity.amount),
            paid_token_x: &U128(need_x),
            paid_token_y: &U128(need_y),
        }
        .emit();
        self.internal_set_user_liquidity(&lpt_id, liquidity);
    }
```

Listing 2.16: contracts/dcl/src/user_liquidity.rs

```
    pub fn merge_liquidity(
        &mut self,
        lpt_id: LptId,
        lpt_id_list: Vec<LptId>
    ) {
        self.assert_contract_running();
        require!(lpt_id_list.len() > 0, E216_INVALID_LPT_LIST);
        let user_id = env::predecessor_account_id();
        let mut retain_liquidity = self.internal_unwrap_user_liquidity(&lpt_id);
        require!(!retain_liquidity.is_mining(), E218_USER_LIQUIDITY_IS_MINING);
        require!(retain_liquidity.owner_id == user_id, E215_NOT_LIQUIDITY_OWNER);
        let mut pool = self.internal_unwrap_pool(&retain_liquidity.pool_id);
        self.assert_no_frozen_tokens(&[pool.token_x.clone(), pool.token_y.clone()]);

        let mut remove_token_x = 0;
        let mut remove_token_y = 0;
        let mut remove_fee_x = 0;
        let mut remove_fee_y = 0;

        let mut merge_lpt_ids = String::new();
        for item in lpt_id_list.iter() {
           merge_lpt_ids = format!("{}{}{}", merge_lpt_ids, if merge_lpt_ids.is_empty() { "" }
                else { "," }, item);
           let user = self.internal_unwrap_user(&user_id);
           let mut liquidity = self.internal_unwrap_user_liquidity(item);
           require!(item != &lpt_id &&
               liquidity.owner_id == retain_liquidity.owner_id &&
               liquidity.pool_id == retain_liquidity.pool_id &&
               liquidity.left_point == retain_liquidity.left_point &&
               liquidity.right_point == retain_liquidity.right_point &&
               !liquidity.is_mining(), E216_INVALID_LPT_LIST);

           let (remove_x, remove_y, acc_fee_x_in_128, acc_fee_y_in_128) =
               pool.internal_remove_liquidity(liquidity.amount, liquidity.left_point,
               liquidity.right_point, 0, 0);

           liquidity.get_unclaimed_fee(acc_fee_x_in_128, acc_fee_y_in_128);
           let fee_x = liquidity.unclaimed_fee_x.unwrap_or(U128(0)).0;
           let fee_y = liquidity.unclaimed_fee_y.unwrap_or(U128(0)).0;

           remove_token_x += remove_x;
           remove_token_y += remove_y;
           remove_fee_x += fee_x;
           remove_fee_y += fee_y;

           pool.total_liquidity -= liquidity.amount;
           pool.total_x -= remove_x + fee_x;
           pool.total_y -= remove_y + fee_y;
           self.internal_burn_liquidity(user, &liquidity);
        }

        let (new_liquidity, need_x, need_y, acc_fee_x_in_128, acc_fee_y_in_128) =
           pool.internal_add_liquidity(retain_liquidity.left_point, retain_liquidity.right_point,
                remove_token_x, remove_token_y, 0, 0);
        retain_liquidity.get_unclaimed_fee(acc_fee_x_in_128, acc_fee_y_in_128);
        let new_fee_x = retain_liquidity.unclaimed_fee_x.unwrap_or(U128(0)).0;
        let new_fee_y = retain_liquidity.unclaimed_fee_y.unwrap_or(U128(0)).0;

        pool.total_liquidity += new_liquidity;
        pool.total_x += need_x;
        pool.total_y += need_y;
        pool.total_x -= new_fee_x;
        pool.total_y -= new_fee_y;

        let refund_x = remove_token_x - need_x + new_fee_x + remove_fee_x;
        let refund_y = remove_token_y - need_y + new_fee_y + remove_fee_y;

        if refund_x > 0{
           if pool.token_x == self.data().wnear_id {
               self.process_near_transfer(&user_id, refund_x);
           } else {
               self.process_ft_transfer(&user_id, &pool.token_x, refund_x);
           }
        }
        if refund_y > 0{
            if pool.token_y == self.data().wnear_id {
                self.process_near_transfer(&user_id, refund_y);
            } else {
                self.process_ft_transfer(&user_id, &pool.token_y, refund_y);
            }
        }

        retain_liquidity.amount += new_liquidity;
        retain_liquidity.last_fee_scale_x_128 = acc_fee_x_in_128;
        retain_liquidity.last_fee_scale_y_128 = acc_fee_y_in_128;

        self.internal_set_pool(&retain_liquidity.pool_id, pool);
        Event::LiquidityMerge {
            lpt_id: &lpt_id,
            merge_lpt_ids: &merge_lpt_ids,
            owner_id: &user_id,
            pool_id: &retain_liquidity.pool_id,
            left_point: &retain_liquidity.left_point,
            right_point: &retain_liquidity.right_point,
            added_amount: &U128(new_liquidity),
            cur_amount: &U128(retain_liquidity.amount),
            paid_token_x: &U128(need_x),
            paid_token_y: &U128(need_y),
        }
        .emit();
        self.internal_set_user_liquidity(&lpt_id, retain_liquidity);
    }
```

Listing 2.17: contracts/dcl/src/user_liquidity.rs

```
    pub fn remove_liquidity(
        &mut self,
        lpt_id: LptId,
        amount: U128,
        min_amount_x: U128,
        min_amount_y: U128,
    ) -> (U128, U128) {
        self.assert_contract_running();
        let user_id = env::predecessor_account_id();
        let user = self.internal_unwrap_user(&user_id);
        let mut liquidity = self.internal_unwrap_user_liquidity(&lpt_id);
        require!(user_id == liquidity.owner_id, E215_NOT_LIQUIDITY_OWNER);
        let mut pool = self.internal_unwrap_pool(&liquidity.pool_id);
        self.assert_no_frozen_tokens(&[pool.token_x.clone(), pool.token_y.clone()]);

        let remove_liquidity = if amount.0 < liquidity.amount { amount.0 } else { liquidity.amount
            };
        if remove_liquidity > 0 {
            require!(!liquidity.is_mining(), E218_USER_LIQUIDITY_IS_MINING);
        }
        let (remove_x, remove_y, acc_fee_x_in_128, acc_fee_y_in_128) = pool.
            internal_remove_liquidity(remove_liquidity, liquidity.left_point, liquidity.
            right_point, min_amount_x.0, min_amount_y.0);
        liquidity.get_unclaimed_fee(acc_fee_x_in_128, acc_fee_y_in_128);

        let new_fee_x = liquidity.unclaimed_fee_x.unwrap_or(U128(0)).0;
        let new_fee_y = liquidity.unclaimed_fee_y.unwrap_or(U128(0)).0;

        liquidity.amount -= remove_liquidity;

        let refund_x = remove_x + new_fee_x;
        let refund_y = remove_y + new_fee_y;
        if refund_x > 0{
            if pool.token_x == self.data().wnear_id {
                self.process_near_transfer(&user_id, refund_x);
            } else {
                self.process_ft_transfer(&user_id, &pool.token_x, refund_x);
            }
        }
        if refund_y > 0{
            if pool.token_y == self.data().wnear_id {
                self.process_near_transfer(&user_id, refund_y);
            } else {
                self.process_ft_transfer(&user_id, &pool.token_y, refund_y);
            }
        }

        pool.total_liquidity -= remove_liquidity;
        pool.total_x -= refund_x;
        pool.total_y -= refund_y;

        self.internal_set_pool(&liquidity.pool_id, pool);

        Event::LiquidityRemoved {
            lpt_id: &lpt_id,
            owner_id: &user_id,
            pool_id: &liquidity.pool_id,
            left_point: &liquidity.left_point,
            right_point: &liquidity.right_point,
            removed_amount: &U128(remove_liquidity),
            cur_amount: &U128(liquidity.amount),
            refund_token_x: &U128(refund_x),
            refund_token_y: &U128(refund_y),
        }
        .emit();

        if liquidity.amount > 0 {
            liquidity.last_fee_scale_x_128 = acc_fee_x_in_128;
            liquidity.last_fee_scale_y_128 = acc_fee_y_in_128;
            self.internal_set_user(&user.user_id.clone(), user);
            self.internal_set_user_liquidity(&lpt_id, liquidity);
        } else {
            self.internal_burn_liquidity(user, &liquidity);
        }

        (refund_x.into(), refund_y.into())
    }
```

Listing 2.18: contracts/dcl/src/user_liquidity.rs

Impact The whole contract instead of affected pools has to be paused in case of emergency.

Suggestion Implement `assert_pool_running()` in above functions.

Feedback from the Project It's a design purpose that we only hold any token exchange (all actions that swap involves) when pausing a pool. So, add/remove order actions would be still active in that case.

2.2.10 Liquidity on Endpoint Processed Before the Limit Order

Severity Low

Status Fixed in `Version 3`

Introduced by `Version 1`

Description Function `internal_x_swap_y_desire_y()` is designed to swap `token_x` to the desired amount of `token_y`. During the swapping process, the liquidity should be processed after the limit order. However, when the point stops at the `next_point`, which is an endpoint, and the amount of `token_x` is not FULLY swapped, the liquidity may be used up while the order is not processed. This is inconsistent with the original design.

```
    pub fn internal_x_swap_y_desire_y(&mut self, protocol_fee_rate: u32, desire_y: u128,
        low_boundary_point: i32, is_quote: bool) -> (u128, u128, bool) {
        require!(desire_y > 0, E205_INVALID_DESIRE_AMOUNT);
        let boundary_point = std::cmp::max(low_boundary_point, LEFT_MOST_POINT);
        let mut is_finished = false;
        let mut amount_x = 0;
        let mut amount_y = 0;
        let mut desire_y = desire_y;
        let mut current_order_or_endpt = self.point_info.get_point_type_value(self.current_point,
            self.point_delta);
        while boundary_point <= self.current_point && !is_finished {
            if current_order_or_endpt & 2 > 0 {
                // process limit order
                let mut point_data = self.point_info.0.get(&self.current_point).unwrap();
                let mut order_data = point_data.order_data.take().unwrap();
                let process_ret = self.process_limit_order_y_desire_y(protocol_fee_rate, &mut
                    order_data, desire_y);
                is_finished = process_ret.0;
                (desire_y, amount_x, amount_y) = (if desire_y <= process_ret.2 { 0 } else {
                    desire_y - process_ret.2 }, amount_x + process_ret.1, amount_y + process_ret.2)
                    ;

                self.update_point_order(&mut point_data, order_data, is_quote);

                if is_finished {
                    break;
                }
            }

        let search_start = self.current_point - 1;

        if current_order_or_endpt & 1 > 0 {
            let process_ret = self.process_liquidity_y_desire_y(protocol_fee_rate, desire_y, self.current_point);
            is_finished = process_ret.0;
            (desire_y, amount_x, amount_y) = (desire_y - std::cmp::min(desire_y, process_ret.2), amount_x+process_ret.1, amount_y+process_ret.2);

            if !is_finished {
                self.pass_endpoint(self.current_point, is_quote, true);
                // move one step to the left
                self.current_point -= 1;
                self.sqrt_price_96 = get_sqrt_price(self.current_point);
                self.liquidity_x = 0;
            }
        }

        if is_finished || self.current_point < boundary_point {
            break;
        }

        let next_pt= match self.slot_bitmap.get_nearest_left_valued_slot(search_start, self.point_delta, boundary_point / self.point_delta){
            Some(point) => {
                if point < boundary_point {
                    boundary_point
                } else {
                    point
                }
             },
            None => { boundary_point }
        };
        let next_val = self.point_info.get_point_type_value(next_pt, self.point_delta);

        if self.liquidity == 0 {
            // no liquidity in the range [next_pt, st.currentPoint)
            self.current_point = next_pt;
            self.sqrt_price_96 = get_sqrt_price(self.current_point);
            current_order_or_endpt = next_val;
        } else {
            let process_ret = self.process_liquidity_y_desire_y(protocol_fee_rate, desire_y, next_pt);
            is_finished = process_ret.0;
            (desire_y, amount_x, amount_y) = (desire_y - std::cmp::min(desire_y, process_ret.2), amount_x+process_ret.1, amount_y+process_ret.2);

            if self.current_point == next_pt {
                current_order_or_endpt = next_val;
            } else {
                current_order_or_endpt = 0;
            }
        }
        if self.current_point <= boundary_point {
            break;
        }
    }
    (amount_x, amount_y, is_finished)
}
```

Listing 2.19: contracts/dcl/src/pool.rs

Impact Liquidity on the endpoint may be swapped out before the limit order on the same endpoint.

Suggestion Process the `liquidity_y` that ranges from the `current_point` to the `next_point+1` first, if there're still some `token_x` left, move to the `next_point`, and handle the limit order before the liquidity on the point.

2.2.11 Lack of Check on Repeated Liquidity in internal_check_remove_liquidity_infos

Severity High

Status Fixed in `Version 5`

Introduced by `Version 4`

Description The function `batch_remove_liquidity()` and `batch_update_liquidity()` are used to remove multiple `liquidity`s of the user in one transaction. The `liquidity` to be removed is passed by the parameter `remove_liquidity_infos`, and validated in the function `internal_check_remove_liquidity_infos()`.

However, if the parameter contains a repeated `liquidity` that is not in the status of mining, the validation will be bypassed, and the repeated `liquidity` will be pushed to the vector `liquiditys`. As a result, every duplicated `liquidity` to be removed will actually be removed only once, while users will be able to receive returned assets corresponding to all these removed `liquidity`s.

```
pub fn internal_check_remove_liquidity_infos(
    &self,
    user: &mut User,
    liquiditys: &mut Vec<UserLiquidity>,
    pool_cache: &mut HashMap<String, Pool>,
    remove_liquidity_infos: &Vec<RemoveLiquidityInfo>,
) -> HashMap<MftId, u128> {
    let mut remove_mft_details = HashMap::new();
    remove_liquidity_infos.iter().for_each(|remove_liquidity_info| {
        let mut liquidity = self.internal_unwrap_user_liquidity(&remove_liquidity_info.lpt_id);
        require!(user.user_id == liquidity.owner_id, E215_NOT_LIQUIDITY_OWNER);
        if remove_liquidity_info.amount.0 > 0 {
            if liquidity.is_mining() {
                if user.mft_assets.get(&liquidity.mft_id).unwrap_or_default() >= liquidity.
                    v_liquidity {
                    user.sub_mft_asset(&liquidity.mft_id, liquidity.v_liquidity);
                    remove_mft_details.entry(liquidity.mft_id).and_modify(|v| *v += liquidity.
                        v_liquidity).or_insert(liquidity.v_liquidity);
                    liquidity.mft_id = String::new();
                    liquidity.v_liquidity = 0;
                }else {
                    env::panic_str(E218_USER_LIQUIDITY_IS_MINING);
                }
            }
        }
        if !pool_cache.contains_key(&liquidity.pool_id) {
            let pool = self.internal_unwrap_pool(&liquidity.pool_id);
            self.assert_no_frozen_tokens(&[pool.token_x.clone(), pool.token_y.clone()]);
            pool_cache.insert(liquidity.pool_id.clone(), pool);
        }
        liquiditys.push(liquidity);
    });
    remove_mft_details
}
```

Listing 2.20: contracts/dcl/src/dcl/user_liquidity. rs

Impact Repeated `liquidity` in the function `batch_remove_liquidity()` and `batch_update_liquidity()` may drain the pool.

Suggestion Check the `liquidity` to be removed accordingly.

2.2.12 Unchecked Received Token in internal_add_order

Severity High

Status Fixed in `Version 5`

Introduced by `Version 1`

Description The function `internal_add_order()` enables users to add orders to a specific pool. Users can indicate which token they would like to add with the specific parameter `token_id`. However, the function does not verify the `token_id` properly. When `token_id` does not match the pool's `token_x`, it is automatically assumed to be the `token_y`, which is incorrect. The `token_id` parameter is entirely under user control, and it could be neither `token_x` nor `token_y`, but instead, a spurious and worthless token.

```
    pub fn internal_add_order(
        &mut self,
        client_id: String,
        user_id: &AccountId,
        token_id: &AccountId,
        amount: Balance,
        pool_id: &PoolId,
        point: i32,
        buy_token: &AccountId,
        swapped_amount: Balance,
        swap_earn_amount: Balance,
    ) -> OrderId {
        let mut pool = self.internal_get_pool(pool_id).unwrap();
        self.assert_no_frozen_tokens(&[pool.token_x.clone(), pool.token_y.clone()]);
        require!(point % pool.point_delta as i32 == 0, E202_ILLEGAL_POINT);
        require!(client_id.len() <= MAX_USER_ORDER_CLIENT_ID_LEN, E306_INVALID_CLIENT_ID);
        require!(amount - swapped_amount > 0, E307_INVALID_SELLING_AMOUNT);


        let mut user = self.internal_unwrap_user(user_id);
        let order_key = gen_user_order_key(pool_id, point);
        require!(
            user.order_keys.get(&order_key).is_none(),
            E301_ACTIVE_ORDER_ALREADY_EXIST
        );


        let global_config = self.internal_get_global_config();
        require!(user.get_available_slots(global_config.storage_price_per_slot, global_config.
             storage_for_asset) > 0, E107_NOT_ENOUGH_STORAGE_FOR_SLOTS);


        let mut point_data = pool.point_info.0.get(&point).unwrap_or_default();
        let prev_active_order = point_data.has_active_order();
        let mut point_order: OrderData = point_data.order_data.unwrap_or_default();


        let order_id = gen_order_id(pool_id, &mut self.data_mut().latest_order_id);
        let mut order = UserOrder {
            client_id,
            order_id: order_id.clone(),
            owner_id: user_id.clone(),
            pool_id: pool_id.clone(),
            point,
            sell_token: token_id.clone(),
            buy_token: buy_token.clone(),
            original_deposit_amount: amount,
            swap_earn_amount,
            original_amount: amount - swapped_amount,
            created_at: env::block_timestamp(),
            last_acc_earn: U256::zero(),
            remain_amount: amount - swapped_amount,
            cancel_amount: 0_u128,
            bought_amount: 0_u128,
            unclaimed_amount: None,
        };


        let (token_x, token_y, _) = pool_id.parse_pool_id();
        if token_x == (*token_id) {
            require!(buy_token == &token_y, E303_ILLEGAL_BUY_TOKEN);
            require!(point >= pool.current_point, E202_ILLEGAL_POINT); // greater or equal to
                current point
            require!(point <= RIGHT_MOST_POINT, E202_ILLEGAL_POINT);
            order.last_acc_earn = point_order.acc_earn_y;
            point_order.selling_x += amount - swapped_amount;
            pool.total_x += amount - swapped_amount;
            pool.total_order_x += amount - swapped_amount;
        } else {
            require!(buy_token == &token_x, E303_ILLEGAL_BUY_TOKEN);
            require!(point <= pool.current_point, E202_ILLEGAL_POINT); // less or equal to current
                point
            require!(point >= LEFT_MOST_POINT, E202_ILLEGAL_POINT);
            order.last_acc_earn = point_order.acc_earn_x;
            point_order.selling_y += amount - swapped_amount;
            pool.total_y += amount - swapped_amount;
            pool.total_order_y += amount - swapped_amount;
        }
        point_order.user_order_count += 1;
        // update order
        user.order_keys.insert(&order_key, &order.order_id);
        self.internal_set_user(user_id, user);


        // update pool info
        point_data.order_data = Some(point_order);
        pool.point_info.0.insert(&point, &point_data);
        if !prev_active_order && !point_data.has_active_liquidity() {
            pool.slot_bitmap.set_one(point, pool.point_delta);
        }
        self.internal_set_pool(pool_id, pool);


        Event::OrderAdded {
            order_id: &order.order_id,
            created_at: &U64(env::block_timestamp()),
            owner_id: &order.owner_id,
            pool_id: &order.pool_id,
            point: &order.point,
            sell_token: &order.sell_token,
            buy_token: &order.buy_token,
            original_amount: &U128(order.original_amount),
            original_deposit_amount: &U128(order.original_deposit_amount),
            swap_earn_amount: &U128(order.swap_earn_amount),
        }
        .emit();
        self.internal_set_user_order(&order_id, order);


        order_id
    }
```

Listing 2.21: contracts/dcl/src/dcl/user_order.rs

Impact Users can add orders to the liquidity pool with no cost, except for the gas fee, and drain the pool after the swap.

Suggestion Ensure that `token_id` is either `token_x` or `token_y`.

2.3 Additional Recommendation

2.3.1 Potential Elastic Supply Token Problem

Status Confirmed

Introduced by Version 1

Description Elastic supply tokens could dynamically adjust their price, supply, user's balance, etc. For example, inflation tokens, deflation tokens, rebasing tokens, etc.

In the current contract implementation, elastic supply tokens are not supported. If the token is a deflation token, there will be a difference between the recorded amount of transferred tokens to this smart contract (as a parameter of function `ft_on_transfer()`) and the actual number of transferred tokens (the token smart contract itself). That's because the token smart contract will burn a small number of tokens.

Suggestion I Do not add elastic supply tokens to the whitelist.

2.3.2 Potential Centralization Problem

Status Confirmed

Introduced by Version 1

Description This project has potential centralization problems. The `ContractData.owner_id` has the privilege to configure several system parameters (e.g., the `ContractData.protocol_fee_rate`) and pause or resume the contract & pools.

Suggestion I Introducing a decentralization design in the contract is recommended, such as a multi-signature or a public DAO.

2.3.3 Redundant Code

Status Fixed in Version 2

Introduced by Version 1

Description In function `update_endpoint()`, if the signed integer `liquidity_data` is checked to be greater than zero, the `liquid_acc_after` will never be less than or equal to the `liquid_acc_before`. Therefore, it is not necessary to have the check in line 162. Similarly, the check in line 169 is also redundant.

```
147    pub fn update_endpoint(
148        &mut self,
149        endpoint: i32,
150        is_left: bool,
151        current_point: i32,
152        liquidity_delta: i128,
153        max_liquidity_per_point: u128,
154        fee_scale_x_128: U256,
155        fee_scale_y_128: U256
156    ) -> bool {
157        let mut point_data = self.0.remove(&endpoint).unwrap_or_default();
158        let mut liquidity_data = point_data.liquidity_data.take().unwrap_or_default();
159        let liquid_acc_before = liquidity_data.liquidity_sum;
160        let liquid_acc_after = if liquidity_delta > 0 {
161            let liquid_acc_after = liquid_acc_before + liquidity_delta as u128;
162            require!(liquid_acc_after > liquid_acc_before);
163            liquid_acc_after
164        } else {
165            let liquid_acc_after = liquid_acc_before - (-liquidity_delta) as u128;
166            require!(liquid_acc_after < liquid_acc_before);
```

```
            liquid_acc_after
        };
        require!(liquid_acc_after <= max_liquidity_per_point, E203_LIQUIDITY_OVERFLOW);
        liquidity_data.liquidity_sum = liquid_acc_after;

        if is_left {
            liquidity_data.liquidity_delta += liquidity_delta;
        } else {
            liquidity_data.liquidity_delta -= liquidity_delta;
        }

        let mut new_or_erase = false;
        if liquid_acc_before == 0 {
            new_or_erase = true;
            if endpoint >= current_point {
                liquidity_data.acc_fee_x_out_128 = fee_scale_x_128;
                liquidity_data.acc_fee_y_out_128 = fee_scale_y_128;
            }
        } else if liquid_acc_after == 0 {
            new_or_erase = true;
        }
        point_data.liquidity_data = Some(liquidity_data);
        self.0.insert(&endpoint, &point_data);
        new_or_erase
    }
```

Listing 2.22: contracts/dcl/src/point_info.rs

Suggestion I It is suggested to remove the redundant checks.

2.3.4 Gas Optimization

Status Fixed in Version 2

Introduced by Version 1

Description In function storage_unregister(), if the user.sponsor_id is the contract itself (env::current-_account_id()), there is no need to send the native NEAR tokens back to itself.

```
    #[payable]
    fn storage_unregister(&mut self, #[allow(unused_variables)] force: Option<bool>) -> bool {
        assert_one_yocto();
        self.assert_contract_running();

        // force option is useless, leave it for compatible consideration.
        // User can NOT unregister if there is still have liquidity, order and asset remain!
        let account_id = env::predecessor_account_id();
        if let Some(user) = self.internal_get_user(&account_id) {
            require!(user.is_empty(), E103_STILL_HAS_REWARD);
            self.data_mut().users.remove(&account_id);
            self.data_mut().user_count -= 1;
            Promise::new(user.sponsor_id).transfer(STORAGE_BALANCE_MIN_BOUND);
            true
        } else {
            false
        }
    }
```

Listing 2.23: cocontracts/dcl/src/storage_impl.rs

Suggestion I If the `sponsor_id` is the contract itself, the transfer of the storage fee is suggested to be skipped.

2.3.5 Unused Code

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Function `gen_liquidity_info_key()` is not used in this contract.

```
pub type LiquidityInfoKey = String;
pub fn gen_liquidity_info_key(left_point: i32, right_point: i32) -> LiquidityInfoKey {
    format!("{}{}{}", left_point, LIQUIDITY_INFO_KEY, right_point)
}
```

Listing 2.24: contracts/dcl/src/utils.rs

Suggestion I It is suggested to remove the unused function `gen_liquidity_info_key()`.

2.3.6 Repeated Variable Assignments

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In function `y_swap_x_range_complete_desire()`, the variable `result.complete_liquidity` is assigned twice in line 694 and line 706.

```
662     /// try to swap from right to left in range [left_point, right_point) with all liquidity used.
663     /// @param liquidity: liquidity of each point in the range
664     /// @param sqrt_price_l_96: sqrt of left point price in 2^96 power
665     /// @param left_point: left point of this range
666     /// @param sqrt_price_r_96: sqrt of right point price in 2^96 power
667     /// @param right_point: right point of this range
668     /// @param desire_x: amount of token X as swap-out
669     /// @return Y2XRangeCompRetDesire
670     pub fn y_swap_x_range_complete_desire(
671         liquidity: u128,
672         sqrt_price_l_96: U256,
673         left_point: i32,
674         sqrt_price_r_96: U256,
675         right_point: i32,
676         desire_x: u128
677     ) -> Y2XRangeCompRetDesire {
678         let mut result = Y2XRangeCompRetDesire::default();
679         let max_x = get_amount_x(liquidity, left_point, right_point, sqrt_price_r_96, sqrt_rate_96
                (), false).as_u128();
680         if max_x <= desire_x {
```

```
681            // maxX <= desireX <= uint128.max
682            result.acquire_x = max_x;
683            result.cost_y = get_amount_y(liquidity, sqrt_price_l_96, sqrt_price_r_96, sqrt_rate_96(), true);
684            result.complete_liquidity = true;
685            return result;
686        }
687
688        let sqrt_price_pr_pl_96 = get_sqrt_price(right_point - left_point);
689        let sqrt_price_pr_m1_96 = sqrt_price_r_96.mul_fraction_floor(pow_96(), sqrt_rate_96());
690        let div = sqrt_price_pr_pl_96 - U256::from(desire_x).mul_fraction_floor(sqrt_price_r_96 - sqrt_price_pr_m1_96, U256::from(liquidity));
691
692        let sqrt_price_loc_96 = sqrt_price_pr_pl_96.mul_fraction_floor(pow_96(), div);
693
694        result.complete_liquidity = false;
695        result.loc_pt = get_log_sqrt_price_floor(sqrt_price_loc_96);
696
697        result.loc_pt = std::cmp::max(left_point, result.loc_pt);
698        result.loc_pt = std::cmp::min(right_point - 1, result.loc_pt);
699        result.sqrt_loc_96 = get_sqrt_price(result.loc_pt);
700
701        if result.loc_pt == left_point {
702            result.acquire_x = 0;
703            result.cost_y = Default::default();
704            return result;
705        }
706        result.complete_liquidity = false;
707        result.acquire_x = std::cmp::min(
708            get_amount_x(liquidity, left_point, result.loc_pt, result.sqrt_loc_96, sqrt_rate_96(), false).as_u128(),
709            desire_x);
710
711        result.cost_y = get_amount_y(liquidity, sqrt_price_l_96, result.sqrt_loc_96, sqrt_rate_96(), true);
712        result
713    }
```

Listing 2.25: contracts/dcl/src/swap_math.rs

Suggestion I It is suggested to remove the repeated assignment of variable `result.complete_liquidity` in line 706.

2.3.7 Incomplete Implementation of Function cancel_order()

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In lines 194-199 of function `cancel_order()`, the logic mentioned in the Todo comments has not been implemented yet.

```
141    /// @param order_id
142    /// @param amount: max cancel amount of selling token
```

```
    /// @return (actual removed sell token, bought token till last update)
    /// Note: cancel_order with 0 amount means claim
    pub fn cancel_order(&mut self, order_id: OrderId, amount: U128) -> (U128, U128) {
        self.assert_contract_running();
        let mut order = self
            .data()
            .user_orders
            .get(&order_id)
            .expect(E304_ORDER_NOT_FOUND);

        let user_id = env::predecessor_account_id();
        require!(order.owner_id == user_id, E300_NOT_ORDER_OWNER);

        let mut pool = self.internal_get_pool(&order.pool_id).unwrap();
        self.assert_pool_running(&pool);
        let mut point_data = pool.point_info.0.get(&order.point).unwrap();
        let mut point_order: OrderData = point_data.order_data.unwrap();

        let earned = internal_update_order(&mut order, &mut point_order);

        // do cancel
        let expected_cancel_amount: Balance = amount.into();
        let actual_cancel_amount = min(expected_cancel_amount, order.remain_amount);
        order.cancel_amount += actual_cancel_amount;
        order.remain_amount -= actual_cancel_amount;

        // update point_data
        if order.is_earn_y() {
            pool.total_x -= actual_cancel_amount;
            pool.total_y -= earned;
            pool.total_order_x -= actual_cancel_amount;
            pool.total_order_y -= earned;
            point_order.selling_x -= actual_cancel_amount;
        } else {
            pool.total_x -= earned;
            pool.total_y -= actual_cancel_amount;
            pool.total_order_x -= earned;
            pool.total_order_y -= actual_cancel_amount;
            point_order.selling_y -= actual_cancel_amount;
        }
        if point_order.selling_x == 0 && point_order.selling_y == 0
        && point_order.earn_y == 0 && point_order.earn_x == 0
        && point_order.earn_y_legacy == 0 && point_order.earn_x_legacy == 0 {
            point_data.order_data = None;
        }

        if point_order.selling_x == 0 && point_order.selling_y == 0 {
            // update slot_bitmap
            if !pool.point_info.is_endpoint(order.point, pool.point_delta) {
                pool.slot_bitmap.set_zero(order.point, pool.point_delta);
            }
            // TODO: will implement remove logic on prod env
            // // see if we can remove point_order
            // if point_order.earn_y == 0 && point_order.earn_x == 0
            // && point_order.earn_y_legacy == 0 && point_order.earn_x_legacy == 0 {
            //     point_data.order_data = None;
            // }
        } else {
            point_data.order_data = Some(point_order);
        }
        pool.point_info.0.insert(&order.point, &point_data);
```

Listing 2.26: contracts/dcl/src/user_order.rs

Suggestion I It is suggested to implement the function `cancel_order()` completely.

2.3.8 Code Optimization

Status Confirmed

Introduced by `Version 1`

Description When a sequence of swap actions is executed in function `internal_swap()`, there is no check on duplicated pools. If a pool with the duplicated pair of `token_x` and `token_y` is involved in the middle of the sequence, the execution of the swap sequence will not fail until it reaches the middle. In this case, the gas is wasted for executing the previous successful swaps.

```
    /// @param account_id
    /// @param pool_ids: all pools participating in swap
    /// @param input_token: the swap-in token, must be in pool_ids[0].tokens
    /// @param input_amount: the amount of swap-in token
    /// @param output_token: the swap-out token, must be in pool_ids[-1].tokens
    /// @param min_output_amount: minimum number of swap-out token to be obtained
    /// @return actual got output token amount
    pub fn internal_swap(
        &mut self,
        account_id: &AccountId,
        pool_ids: Vec<PoolId>,
        input_token: &AccountId,
        input_amount: Balance,
        output_token: &AccountId,
        min_output_amount: Balance,
    ) -> Balance {
        pool_ids.iter().for_each(|pool_id| self.assert_pool_running(&self.internal_unwrap_pool(
            pool_id)));
        let mut pool_record = HashSet::new();
        let protocol_fee_rate = self.data().protocol_fee_rate;
        let (actual_output_token, actual_output_amount) = {
            let mut next_input_token_or_last_output_token = input_token.clone();
            let mut next_input_amount_or_actual_output = input_amount;
            for pool_id in pool_ids {
                let mut pool = self.internal_unwrap_pool(&pool_id);
                let is_not_exist = pool_record.insert(format!("{}|{}", pool.token_x, pool.token_y))
                    ;
                require!(is_not_exist, E206_DUPLICATE_POOL);
                if next_input_token_or_last_output_token.eq(&pool.token_x) {
                    let (actual_cost, out_amount, is_finished) =
                        pool.internal_x_swap_y(protocol_fee_rate, next_input_amount_or_actual_output, -799999, false);
                    if !is_finished {
                        env::panic_str(&format!("ERR_TOKEN_{}_NOT_ENOUGH", pool.token_y.to_string().to_uppercase()));
                    }

                    pool.total_x += actual_cost;
                    pool.total_y -= out_amount;
                    pool.volume_x_in += U256::from(actual_cost);
                    pool.volume_y_out += U256::from(out_amount);

                    next_input_token_or_last_output_token = pool.token_y.clone();
                    next_input_amount_or_actual_output = out_amount;
                } else if next_input_token_or_last_output_token.eq(&pool.token_y) {
                    let (actual_cost, out_amount, is_finished) =
                        pool.internal_y_swap_x(protocol_fee_rate, next_input_amount_or_actual_output, 799999, false);
                    if !is_finished {
                        env::panic_str(&format!("ERR_TOKEN_{}_NOT_ENOUGH", pool.token_x.to_string().to_uppercase()));
                    }

                    pool.total_y += actual_cost;
                    pool.total_x -= out_amount;
                    pool.volume_y_in += U256::from(actual_cost);
                    pool.volume_x_out += U256::from(out_amount);

                    next_input_token_or_last_output_token = pool.token_x.clone();
                    next_input_amount_or_actual_output = out_amount;
                } else {
                    env::panic_str(E404_INVALID_POOL_IDS);
                }
                self.internal_set_pool(&pool_id, pool);
            }
            (
                next_input_token_or_last_output_token,
                next_input_amount_or_actual_output,
            )
        };

        require!(output_token == &actual_output_token, E212_INVALID_OUTPUT_TOKEN);
        require!(actual_output_amount >= min_output_amount, E204_SLIPPAGE_ERR);

        if actual_output_amount > 0 {
            if output_token == &self.data().wnear_id {
                self.process_near_transfer(account_id, actual_output_amount);
            } else {
                self.process_ft_transfer(account_id, output_token, actual_output_amount);
            }
        }
        Event::Swap {
            swapper: account_id,
            token_in: input_token,
            token_out: output_token,
            amount_in: &U128(input_amount),
            amount_out: &U128(actual_output_amount),
        }
        .emit();
        actual_output_amount
    }
```

Listing 2.27: contracts/dcl/src/swap.rs

Suggestion I Check all the pools listed in `pool_ids` before the swap to ensure no duplicate pools exist.

2.3.9 Unsupported Token Frozen List

Status Fixed in `Version 2`

Introduced by `Version 1`

Description According to the current management of the contract, the contract owner (perhaps a public `DAO`) can not directly freeze a specified token for some potential emergency.

Suggestion I It is suggested to introduce a feature that can manage the status of tokens as frozen or unfrozen independently.

2.3.10 Redundant Clone in nft_transfer_call()

Status Fixed in `Version 3`

Introduced by `Version 1`

Description In function `nft_transfer_call()`, the input parameters `prev_owner`, `receiver_id`, and `token_id` will not be used in the function `nft_transfer_call()` after the callback function (i.e., `nft_resolve_transfer()`). In this case, there is no need to clone them for saving gas.

```
    #[payable]
    fn nft_transfer_call(
        &mut self,
        receiver_id: AccountId,
        token_id: TokenId,
        approval_id: Option<u64>,
        memo: Option<String>,
        msg: String,
    ) -> PromiseOrValue<bool> {
        assert_one_yocto();
        require!(
            env::prepaid_gas() > GAS_FOR_NFT_TRANSFER_CALL,
            E501_MORE_GAS_IS_REQUIRED
        );
        self.assert_contract_running();
        let sender_id = env::predecessor_account_id();
        let (prev_owner, old_approvals) = self.internal_transfer(&token_id, &sender_id, &
             receiver_id, approval_id, memo);
        // Initiating receiver's call and the callback
        ext_receiver::ext(receiver_id.clone())
            .with_attached_deposit(NO_DEPOSIT)
            .with_static_gas(env::prepaid_gas() - GAS_FOR_NFT_TRANSFER_CALL)
            .nft_on_transfer(sender_id.clone(), prev_owner.clone(), token_id.clone(), msg)
            .then(
                Self::ext(env::current_account_id())
                    .with_static_gas(GAS_FOR_RESOLVE_TRANSFER)
                    .nft_resolve_transfer(
                        prev_owner.clone(),
                        receiver_id.clone(),
                        token_id.clone(),
                        old_approvals,
                    ),
            )
            .into()
    }
```

Listing 2.28: contracts/dcl/src/nft.rs

Suggestion I Remove the function `clone()` for the above mentioned parameters.

2.3.11 Redundant Information in MftId

Status Confirmed

Introduced by `Version 2`

Description Function `gen_mft_id()` is used to generate the `MftId` for the corresponding `mft` token, which consists of the `FarmingType`, `pool_id`, `left_point` and `right_point`. However, the `FarmingType` already contains the `left_point` and the `right_point` of the `mft`, which is duplicate.

```
pub type MftId = String;
pub fn gen_mft_id(pool_id: &PoolId, farming_type: &FarmingType) -> MftId {
    match farming_type{
        FarmingType::FixRange { left_point, right_point } => {
            format!(":{}{}{}{}{}{}{}", near_sdk::serde_json::to_string(farming_type).unwrap(),
                MFT_ID_BREAK, pool_id, MFT_ID_BREAK, left_point, MFT_ID_BREAK, right_point)
        }
    }
}
```

Listing 2.29: contracts/dcl/src/utils.rs

Suggestion I Delete the redundant information (i.e., `left_point` and `right_point`) in `MftId`.

Feedback from the Project This redundancy is designed to be like this, according to frontend development requests.

2.3.12 Lack of Check on Duplicate Tokens in Frozen List

Status Confirmed

Introduced by `Version 1`

Description The owner and operators can freeze tokens via the function extend_frozenlist_tokens(). However, the duplicate tokens in the input are not checked. In this case, the token which is supposed to be added in the list may be omitted.

```
#[payable]
pub fn extend_frozenlist_tokens(&mut self, tokens: Vec<AccountId>) {
    assert_one_yocto();
    require!(self.is_owner_or_operators(), E002_NOT_ALLOWED);
    for token in tokens {
        self.data_mut().frozenlist.insert(&token);
    }
}
```

Listing 2.30: contracts/dcl/src/management.rs

Suggestion I Check the return value of function insert() in the for loop.

Feedback from the Project As a management interface, operators would check the execution result and corresponding effect to make sure the right tokens are correctly set.

2.3.13 Potential Failure of NEAR Transfer

Status Confirmed

Introduced by Version 1

Description In the callback function callback_post_withdraw_near(), if the PromiseResult is checked as Successful, the contract will transfer NEARs to the user. However, the transfer may fail due to the unregistration of the user's NEAR account.

```
#[private]
pub fn callback_post_withdraw_near(
    &mut self,
    user_id: AccountId,
    amount: U128,
) -> bool {
    require!(
        env::promise_results_count() == 1,
        E001_PROMISE_RESULT_COUNT_INVALID
    );
    let amount: Balance = amount.into();
    match env::promise_result(0) {
        PromiseResult::NotReady => unreachable!(),
        PromiseResult::Successful(_) => {
            Promise::new(user_id).transfer(amount);
            true
        }
        PromiseResult::Failed => {
            // This reverts the changes from withdraw function.
            if let Some(mut user) = self.internal_get_user(&user_id) {
                user.add_asset_uncheck(&self.data().wnear_id, amount);
                self.internal_set_user(&user_id, user);

                Event::Lostfound {
                    user: &user_id,
                    token: &self.data().wnear_id,
                    amount: &U128(amount),
                    locked: &false,
                }
                .emit();
            } else {
                Event::Lostfound {
                    user: &user_id,
                    token: &self.data().wnear_id,
                    amount: &U128(amount),
                    locked: &true,
                }
                .emit();
            }
            false
        }
    }
}
```

Listing 2.31: contracts/dcl/src/user_asset.rs

Suggestion I It's suggested to print a log for the potential failure, which is similar to the implementation when `PromiseResult` returned as `Failed`.

Feedback from the Project Although it is a really rare condition, if the account was deleted before transfer, it could be taken as a donation cause even if we record this kind of transfer failure, we could not re-transfer it when the account is back online. As we are unable to tell if this new account owner is the one before.

2.3.14 Skipped Transfer in Function storage_deposit and storage_deposit

Status Fixed in `Version 5`

Introduced by `Version 4`

Description In function `storage_unregister()`, NEAR transfer is skipped if the `sponsor_id` matches `env::current_account_id`. This helps to save gas. However, this check is not present in the function `storage_withdraw()` and `storage_deposit()`.

```
#[payable]
fn storage_deposit(
    &mut self,
    account_id: Option<AccountId>,
    registration_only: Option<bool>,
) -> StorageBalance {
    self.assert_contract_running();

    let amount = env::attached_deposit();
    let account_id = account_id.unwrap_or_else(env::predecessor_account_id);
    let caller_id = env::predecessor_account_id();
    let already_registered = self.data().users.contains_key(&account_id);
    let registration_only = registration_only.unwrap_or_default();
```

```
        if amount < STORAGE_BALANCE_MIN_BOUND && !already_registered {
            env::panic_str(E102_INSUFFICIENT_STORAGE);
        }

        if already_registered {
            if amount > 0 {
                let mut user = self.internal_unwrap_user(&account_id);

                if caller_id == account_id && account_id != user.sponsor_id {
                    require!(amount >= user.locked_near_for_storage);
                    Promise::new(user.sponsor_id).transfer(user.locked_near_for_storage);
                    user.sponsor_id = caller_id;
                    user.locked_near_for_storage = amount;
                } else {
                    user.locked_near_for_storage += amount;
                }
                Event::AppendUserStorage {
                    operator: &env::predecessor_account_id(),
                    user: &account_id,
                    amount: &U128(amount),
                }.emit();
                self.internal_set_user(&account_id, user);
            }
        } else {
            let actual_amount =
            if registration_only {
                self.internal_set_user(&account_id, User::new(&account_id, &caller_id,
                     STORAGE_BALANCE_MIN_BOUND));
                let refund = amount - STORAGE_BALANCE_MIN_BOUND;
                if refund > 0 {
                    Promise::new(env::predecessor_account_id()).transfer(refund);
                }
                STORAGE_BALANCE_MIN_BOUND
            } else {
                self.internal_set_user(&account_id, User::new(&account_id, &caller_id, amount));
                amount
            };
            self.data_mut().user_count += 1;
            Event::InitUserStorage {
                operator: &env::predecessor_account_id(),
                user: &account_id,
                amount: &U128(actual_amount),
            }.emit();
        }
        self.storage_balance_of(account_id).unwrap()
    }

    #[payable]
    fn storage_withdraw(
        &mut self,
        amount: Option<U128>,
    ) -> StorageBalance {
        assert_one_yocto();
```

```rust
        self.assert_contract_running();

        let account_id = env::predecessor_account_id();
        let mut user = self.internal_unwrap_user(&account_id);
        let receiver_id = user.sponsor_id.clone();
        let global_config = self.internal_get_global_config();
        let storage_price_per_slot = global_config.storage_price_per_slot;
        let available_slots = user.get_available_slots(storage_price_per_slot, global_config.
            storage_for_asset);

        let max_amount = available_slots as u128 * storage_price_per_slot;
        let withdraw_amount = if let Some(a) = amount {
            if a.0 > max_amount { max_amount } else { a.0 }
        } else {
            max_amount
        };

        user.locked_near_for_storage -= withdraw_amount;

        Event::WithdrawUserStorage {
            operator: &account_id,
            receiver: &receiver_id,
            amount: &U128(withdraw_amount),
            remain: &U128(user.locked_near_for_storage),
        }.emit();

        self.internal_set_user(&account_id, user);

        if withdraw_amount > 0 {
            Promise::new(receiver_id).transfer(withdraw_amount);
        }

        self.storage_balance_of(account_id).unwrap()
    }
```

Listing 2.32: contracts/dcl/src/api/storage_api.rs

Suggestion I Add the corresponding check in the function `storage_withdraw()` and `storage_deposit()`.

2.3.15 Lack of Check on Empty Argument

Status Fixed in `Version 5`

Introduced by `Version 4`

Description The function `set_vip_user()` is used to configure the discounts of swap fees for certain pools of the user. However, if the `discount` is empty, it is meaningless to save the user in the contract.

```rust
    #[payable]
    pub fn set_vip_user(&mut self, user: AccountId, discount: HashMap<PoolId, u32>) {
        assert_one_yocto();
        require!(self.is_owner_or_operators(), E002_NOT_ALLOWED);
        require!(discount.iter().all(|(_, &v)| v as u128 <= BP_DENOM),
            E011_INVALID_VIP_USER_DISCOUNT);
        self.data_mut().vip_users.insert(&user, &discount);
    }
```

Listing 2.33: contracts/dcl/src/api/management.rs

Suggestion I Add a check in the function `set_vip_user()` to verify whether the `discount` is empty.

2.3.16 Spelling Error

Status Fixed in `Version 5`

Introduced by `Version 1`

Description The spelling of these variables is inappropriate.

File	Variable
contracts/dcl/src/errors.rs, line 22	`E104_INSURFFICIENT_DEPOIST`
contracts/dcl/src/errors.rs, line 72	`E505_SNEDER_NOT_APPROVED`

Suggestion I Correct the spelling mistakes.

2.3.17 Redundant Event Emission in View Functions

Status Fixed in `Version 7`

Introduced by `Version 6`

Description Function `internal_add_liquidity` will always emit the `Event::LiquidityAdded` even when `is_view` set as true. While this does not modify the contract's state, it may lead to inaccuracies in off-chain statistics and analytics.

```
    pub fn internal_add_liquidity(
        &self,
        pool: &mut Pool,
        user_id: &AccountId,
        lpt_id: LptId,
        left_point: i32,
        right_point: i32,
        amount_x: U128,
        amount_y: U128,
        min_amount_x: U128,
        min_amount_y: U128,
        is_view: bool
    ) -> (u128, u128, UserLiquidity) {
        let (new_liquidity, need_x, need_y, acc_fee_x_in_128, acc_fee_y_in_128) = pool.
            internal_add_liquidity(left_point, right_point, amount_x.0, amount_y.0, min_amount_x
            .0, min_amount_y.0, is_view);
        let liquidity = UserLiquidity {
            lpt_id: lpt_id.clone(),
            owner_id: user_id.clone(),
            pool_id: pool.pool_id.clone(),
            left_point,
            right_point,
            last_fee_scale_x_128: acc_fee_x_in_128,
            last_fee_scale_y_128: acc_fee_y_in_128,
            amount: new_liquidity,
            mft_id: String::new(),
            v_liquidity: 0,
            unclaimed_fee_x: None,
            unclaimed_fee_y: None,
        };

        pool.total_liquidity += new_liquidity;
        pool.total_x += need_x;
        pool.total_y += need_y;

        Event::LiquidityAdded {
            lpt_id: &lpt_id,
            owner_id: &user_id,
            pool_id: &pool.pool_id,
            left_point: &left_point,
            right_point: &right_point,
            added_amount: &U128(new_liquidity),
            cur_amount: &U128(liquidity.amount),
            paid_token_x: &U128(need_x),
            paid_token_y: &U128(need_y),
        }
        .emit();

        (need_x, need_y, liquidity)
    }
```

Listing 2.34: contracts/dcl/src/dcl/user_liquidity.rs

Suggestion I Avoid emitting `Event::LiquidityAdded` when `is_view` is true.

2.4 Notes

2.4.1 Assumption on the Secure Implementation of Contract Dependencies

Status Confirmed

Introduced by `Version 1`

Description The `Ref_DCL_Contract` is built based on the crates `NEAR-SDK` (version 4.0.0) and `near-contract-standards` (version 4.0.0).

```
use near_contract_standards::non_fungible_token::core::NonFungibleTokenCore;
use near_contract_standards::non_fungible_token::core::NonFungibleTokenResolver;
use near_contract_standards::non_fungible_token::enumeration::NonFungibleTokenEnumeration;
use near_contract_standards::non_fungible_token::events::NftTransfer;
use near_contract_standards::non_fungible_token::metadata::{
    NFTContractMetadata, NonFungibleTokenMetadataProvider, NFT_METADATA_SPEC,
};
use near_contract_standards::non_fungible_token::{Token, TokenId};
```

Listing 2.35: contracts/dcl/src/nft.rs

```
use near_contract_standards::non_fungible_token::approval::NonFungibleTokenApproval;
use near_contract_standards::non_fungible_token::approval::ext_nft_approval_receiver;
use near_contract_standards::non_fungible_token::TokenId;
```

Listing 2.36: contracts/dcl/src/nft_approval.rs

```
use near_contract_standards::storage_management::{
    StorageBalance, StorageBalanceBounds, StorageManagement,
};
```

Listing 2.37: contracts/dcl/src/storage_impl.rs

The required interfaces and the basic functionality listed below are provided in the contract:

* NEP-171 (Non-Fungible Token Core Functionality)
* NEP-178 (Non-Fungible Token Approval Management)
* NEP-181 (Non-Fungible Token Enumeration)
* NEP-177 (Non-Fungible Token Metadata Standard)
* NEP-297 (Events Standard)
* NEP-145 (Storage Management)

In this audit, we assume the standard library provided by NEAR-SDK-RS [1] (i.e., near_contract_standards) has no security issues.

2.4.2 Unsupported Increasement of Selling Tokens for Limit Orders

Status Confirmed

Introduced by Version 1

Description Users can reduce the amount of selling tokens for a specific limit order by invoking the function cancel_order().

```
    /// @param order_id
    /// @param amount: max cancel amount of selling token
    /// @return (actual removed sell token, bought token till last update)
    /// Note: cancel_order with 0 amount means claim
    pub fn cancel_order(&mut self, order_id: OrderId, amount: U128) -> (U128, U128) {
        self.assert_contract_running();
        let mut order = self
            .data()
            .user_orders
            .get(&order_id)
            .expect(E304_ORDER_NOT_FOUND);

        let user_id = env::predecessor_account_id();
        require!(order.owner_id == user_id, E300_NOT_ORDER_OWNER);

        let mut pool = self.internal_get_pool(&order.pool_id).unwrap();
```

[1] https://github.com/near/near-sdk-rs

```
        self.assert_pool_running(&pool);
        let mut point_data = pool.point_info.0.get(&order.point).unwrap();
        let mut point_order: OrderData = point_data.order_data.unwrap();

        let earned = internal_update_order(&mut order, &mut point_order);

        // do cancel
        let expected_cancel_amount: Balance = amount.into();
        let actual_cancel_amount = min(expected_cancel_amount, order.remain_amount);
        order.cancel_amount += actual_cancel_amount;
        order.remain_amount -= actual_cancel_amount;

        // update point_data
        if order.is_earn_y() {
            pool.total_x -= actual_cancel_amount;
            pool.total_y -= earned;
            pool.total_order_x -= actual_cancel_amount;
            pool.total_order_y -= earned;
            point_order.selling_x -= actual_cancel_amount;
        } else {
            pool.total_x -= earned;
            pool.total_y -= actual_cancel_amount;
            pool.total_order_x -= earned;
            pool.total_order_y -= actual_cancel_amount;
            point_order.selling_y -= actual_cancel_amount;
        }
        if point_order.selling_x == 0 && point_order.selling_y == 0
        && point_order.earn_y == 0 && point_order.earn_x == 0
        && point_order.earn_y_legacy == 0 && point_order.earn_x_legacy == 0 {
            point_data.order_data = None;
        }

        if point_order.selling_x == 0 && point_order.selling_y == 0 {
            // update slot_bitmap
            if !pool.point_info.is_endpoint(order.point, pool.point_delta) {
                pool.slot_bitmap.set_zero(order.point, pool.point_delta);
            }
            // TODO: will implement remove logic on prod env
            // // see if we can remove point_order
            // if point_order.earn_y == 0 && point_order.earn_x == 0
            // && point_order.earn_y_legacy == 0 && point_order.earn_x_legacy == 0 {
            //     point_data.order_data = None;
            // }
        } else {
            point_data.order_data = Some(point_order);
        }
        pool.point_info.0.insert(&order.point, &point_data);
```

Listing 2.38: contracts/dcl/src/user_order.rs

However, on the contrary, no function can be used to increase the amount of selling tokens in a limit order.

```
    /// Place order at given point
    /// @param user_id: the owner of this order
    /// @param token_id: the selling token
    /// @param amount: the amount of selling token for this order
    /// @param pool_id: pool of this order
    /// @param buy_token: the token this order want to buy
    /// @return OrderId
    pub fn internal_add_order(
        &mut self,
        user_id: &AccountId,
        token_id: &AccountId,
        amount: Balance,
        pool_id: &PoolId,
        point: i32,
        buy_token: &AccountId,
        swapped_amount: Balance,
        swap_earn_amount: Balance,
    ) -> OrderId {
        let mut pool = self.internal_get_pool(pool_id).unwrap();
        self.assert_pool_running(&pool);
        require!(point % pool.point_delta as i32 == 0, E202_ILLEGAL_POINT);

        let mut user = self.internal_unwrap_user(user_id);
        let order_key = gen_user_order_key(pool_id, point);
        require!(
            user.order_keys.get(&order_key).is_none(),
            E301_ACTIVE_ORDER_ALREADY_EXIST
        );
        require!(
            user.order_keys.len() < DEFAULT_MAX_USER_ACTIVE_ORDER_COUNT,
            E302_USER_ACTIVE_ORDER_NUM_EXCEEDED
        );

        let mut point_data = pool.point_info.0.get(&point).unwrap_or_default();
        let mut point_order: OrderData = point_data.order_data.unwrap_or_default();

        let mut order = UserOrder {
            order_id: gen_order_id(pool_id, &mut self.data_mut().latest_order_id),
            owner_id: user_id.clone(),
            pool_id: pool_id.clone(),
            point,
            sell_token: token_id.clone(),
            buy_token: buy_token.clone(),
            original_deposit_amount: amount,
            swap_earn_amount,
            original_amount: amount - swapped_amount,
            created_at: env::block_timestamp(),
            last_acc_earn: U256::zero(),
            remain_amount: amount - swapped_amount,
            cancel_amount: 0_u128,
            bought_amount: 0_u128,
            unclaimed_amount: None,
        };

        let (token_x, token_y, _) = pool_id.parse();
        if token_x == (*token_id) {
            require!(buy_token == &token_y, E303_ILLEGAL_BUY_TOKEN);
            require!(point >= pool.current_point, E202_ILLEGAL_POINT); // greater or equal to
                current point
            require!(point <= RIGHT_MOST_POINT, E202_ILLEGAL_POINT);
            order.last_acc_earn = point_order.acc_earn_y;
            point_order.selling_x += amount - swapped_amount;
            pool.total_x += amount - swapped_amount;
            pool.total_order_x += amount - swapped_amount;
        } else {
            require!(buy_token == &token_x, E303_ILLEGAL_BUY_TOKEN);
            require!(point <= pool.current_point, E202_ILLEGAL_POINT); // less or equal to current
                point
            require!(point >= LEFT_MOST_POINT, E202_ILLEGAL_POINT);
            order.last_acc_earn = point_order.acc_earn_x;
            point_order.selling_y += amount - swapped_amount;
            pool.total_y += amount - swapped_amount;
            pool.total_order_y += amount - swapped_amount;
        }
        // update order
        user.order_keys.insert(&order_key, &order.order_id);
        self.internal_set_user(user_id, user);
        self.data_mut().user_orders.insert(&order.order_id, &order);

        // update pool info
        point_data.order_data = Some(point_order);
        pool.point_info.0.insert(&point, &point_data);
        pool.slot_bitmap.set_one(point, pool.point_delta);
        self.internal_set_pool(pool_id, pool);

        Event::OrderAdded {
            order_id: &order.order_id,
            created_at: &U64(env::block_timestamp()),
            owner_id: &order.owner_id,
            pool_id: &order.pool_id,
            point: &order.point,
            sell_token: &order.sell_token,
            buy_token: &order.buy_token,
            original_amount: &U128(order.original_amount),
            original_deposit_amount: &U128(order.original_deposit_amount),
            swap_earn_amount: &U128(order.swap_earn_amount),
        }
        .emit();

        order.order_id.clone()
    }
}
```

Listing 2.39: contracts/dcl/src/user_order.rs

2.4.3 Unsupported Deposit of Native NEAR Tokens

Status Confirmed

Introduced by Version 1

Description When processing the wNEAR transfer, the unwrapped native NEAR tokens will be transferred instead of the wNEAR.

```rust
pub fn process_near_transfer(&mut self, user_id: &AccountId, amount: Balance) -> Promise {
ext_wrap_near::ext(self.data().wnear_id.clone())
    .with_attached_deposit(1)
    .with_static_gas(GAS_FOR_NEAR_WITHDRAW)
    .near_withdraw(amount.into())
    .then(
        Self::ext(env::current_account_id())
            .with_static_gas(GAS_FOR_RESOLVE_NEAR_WITHDRAW)
            .callback_post_withdraw_near(
                user_id.clone(),
                amount.into(),
            ),
    )
}
```

Listing 2.40: contracts/dcl/src/user_asset.rs

However, on the contrary, this contract does not accept native NEAR tokens as deposits, which may cause inconvenience to the users.