



# Security Audit

# Report for DeltaTrade

**Date:** August 22, 2024 **Version:** 1.0

**Contact:** [contact@blocksec.com](mailto:contact@blocksec.com)

# Contents

<b>Chapter 1 Introduction</b>	<b>1</b>
1.1 About Target Contracts	1
1.2 Disclaimer	1
1.3 Procedure of Auditing	1
1.3.1 Software Security	2
1.3.2 DeFi Security	2
1.3.3 NFT Security	2
1.3.4 Additional Recommendation	2
1.4 Security Model	3
<b>Chapter 2 Findings</b>	<b>4</b>
2.1 DeFi Security	6
2.1.1 Lack of check in function <code>set_min_deposit()</code>	6
2.1.2 Lack of refunding <code>EXECUTE_DCA_FEE</code> in function <code>execute_dca()</code>	8
2.1.3 Lack of check on <code>DCA</code> status in function <code>close_dca()</code>	9
2.1.4 Lack of check on contract status in function <code>token_storage_deposit()</code>	10
2.1.5 Incorrect check of promise results in function <code>callback_do_withdraw()</code>	11
2.1.6 Inconsistent update time between <code>global_balances_map</code> and token balance	13
2.1.7 Incorrect logic in function <code>withdraw_asset_from_ref()</code>	13
2.1.8 Incorrect check in function <code>internal_execute_buy()</code>	15
2.1.9 Potential user losses due to incorrect swap path	17
2.1.10 Unscaled expo values in <code>Pyth-Oracle</code> integration	19
2.1.11 Potential user losses due to manipulated <code>amount_out</code>	20
2.1.12 Lack of setting static gas in function <code>after_withdraw_near()</code>	21
2.1.13 Lack of depositing storage fee for <code>token_out</code>	22
2.1.14 Lack of lock during the withdrawals from <code>Ref-Exchange</code>	26
2.2 Additional Recommendation	28
2.2.1 Redundant code	28
2.2.2 Standardize owner checks with <code>assert_owner_without_yocto()</code>	30
2.2.3 Lack of setting static gas	32
2.2.4 Lack of check in function <code>token_storage_deposit()</code>	33
2.2.5 Incorrect gas calculation	34
2.3 Note	34
2.3.1 Decision of swap path in function <code>execute_dca()</code>	34
2.3.2 Potential centralization risk	35

## Report Manifest

Item	Description
Client	DeltaTrade
Target	DeltaTrade

## Version History

Version	Date	Description
1.0	August 22, 2024	First release

## Signature

**About BlockSec** BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at [Email](#), [Twitter](#) and [Medium](#).

# Chapter 1 Introduction

## 1.1 About Target Contracts

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

The target of this audit is the code repository of DeltaTrade<sup>1</sup> of DeltaTrade.

The auditing process is iterative. Specifically, we would 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 table. Our audit report is responsible for the code in the initial version ([Version 1](#)), as well as new code (in the following versions) to fix issues in the audit report.

Project	Version	Commit Hash
DeltaTrade	<a href="#">Version 1</a>	<a href="#">d3411c3e01d1d96b5fdeff7b4e82e58dac1c433b</a>
	<a href="#">Version 2</a>	<a href="#">6dbe5a2f95af2273d1ec096e95c33d2e4f74a7b7</a>

## 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.

---

<sup>1</sup><https://github.com/DeltaBotDev/DCA>

- **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 <sup>2</sup> and Common Weakness Enumeration <sup>3</sup>. 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

<b>Impact</b>	<i>High</i>	High	Medium
	<i>Low</i>	Medium	Low
		<i>High</i>	<i>Low</i>
		<b>Likelihood</b>	

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.

<sup>2</sup>[https://owasp.org/www-community/OWASP\\_Risk\\_Rating\\_Methodology](https://owasp.org/www-community/OWASP_Risk_Rating_Methodology)

<sup>3</sup><https://cwe.mitre.org/>

## Chapter 2 Findings

In total, we find **fourteen** potential issues. Besides, we also have **five** recommendations and **two** notes as follows:

- High Risk: 6
- Medium Risk: 4
- Low Risk: 4
- Recommendation: 5
- Note: 2

ID	Severity	Description	Category	Status
1	Low	Lack of check in function <code>set_min_deposit()</code>	DeFi Security	Fixed
2	Low	Lack of refunding <code>EXECUTE_DCA_FEE</code> in function <code>execute_dca()</code>	DeFi Security	Confirmed
3	High	Lack of check on DCA status in function <code>close_dca()</code>	DeFi Security	Fixed
4	Low	Lack of check on contract status in function <code>token_storage_deposit()</code>	DeFi Security	Fixed
5	High	Incorrect check of promise results in function <code>callback_do_withdraw()</code>	DeFi Security	Fixed
6	Medium	Inconsistent update time between <code>global_balances_map</code> and token balance	DeFi Security	Fixed
7	Medium	Incorrect logic in function <code>withdraw_asset_from_ref()</code>	DeFi Security	Fixed
8	Medium	Incorrect check in function <code>internal_execute_buy()</code>	DeFi Security	Fixed
9	High	Potential user losses due to incorrect swap path	DeFi Security	Fixed
10	High	Unscaled expo values in <code>Pyth-Oracle</code> integration	DeFi Security	Fixed
11	High	Potential user losses due to manipulated <code>amount_out</code>	DeFi Security	Fixed
12	Low	Lack of setting static gas in function <code>after_withdraw_near()</code>	DeFi Security	Fixed
13	Medium	Lack of depositing storage fee for <code>token_out</code>	DeFi Security	Fixed
14	High	Lack of lock during the withdrawals from <code>Ref-Exchange</code>	DeFi Security	Fixed
15	-	Redundant code	Recommendation	Confirmed
16	-	Standardize owner checks with <code>assert_owner_without_yocto()</code>	Recommendation	Fixed

17	-	Lack of setting static gas	Recommendation	Fixed
18	-	Lack of check in function <code>token_storage_deposit()</code>	Recommendation	Confirmed
19	-	Incorrect gas calculation	Recommendation	Confirmed
20	-	Decision of swap path in function <code>execute_dca()</code>	Note	
21	-	Potential centralization risk	Note	



The details are provided in the following sections.

## 2.1 DeFi Security

### 2.1.1 Lack of check in function `set_min_deposit()`

**Severity** Low

**Status** Fixed in [Version 2](#)

**Introduced by** [Version 1](#)

**Description** Function `set_min_deposit()` can be used by the owner to set the minimum deposit for a specified token. Consequently, the token is inserted into `deposit_limit_map`, which is used to verify whether these tokens are whitelisted during the [DCA](#) creation and storage fee deposit process. However, these tokens may not be registered through the function `register_pair()` and are not in the `global_balances_map`. In this case, this check did not function as intended.

```
112 pub fn set_min_deposit(&mut self, token: AccountId, min_deposit: U128) {
113     require!(env::predecessor_account_id() == self.owner_id, ERR_NOT_ALLOWED);
114     require!(env::attached_deposit() == DEFAULT_CONFIG_SET_STORAGE_FEE, LESS_STORAGE_FEE);
115     self.deposit_limit_map.insert(&token, &min_deposit);
116 }
```

**Listing 2.1:** `dca_owner.rs`

```
81 pub fn register_pair(&mut self, token_a: AccountId, token_b: AccountId, token_a_min_deposit:
    U128, token_b_min_deposit: U128, token_a_oracle_id_op: Option<String>,
    token_b_oracle_id_op: Option<String>, path: Vec<Pool>) {
82     require!(env::predecessor_account_id() == self.owner_id, ERR_NOT_ALLOWED);
83     require!(env::attached_deposit() == REGISTER_PAIR_STORAGE_FEE * 2, LESS_STORAGE_FEE);
84     require!(token_a == path.get(0).unwrap().token_in && token_b == path.get(path.len() - 1).
        unwrap().token_out, INVALID_TOKEN);
85
86
87     let pair_key = self.internal_get_pair_key(&token_a, &token_b);
88     // record Pair pool id
89     self.recorded_pair_path.insert(&pair_key, &path);
90
91
92     self.deposit_limit_map.insert(&token_a, &token_a_min_deposit);
93     self.deposit_limit_map.insert(&token_b, &token_b_min_deposit);
94
95
96     self.internal_increase_global_asset(&token_a, &U128::from(0));
97     self.internal_increase_global_asset(&token_b, &U128::from(0));
98
99
100    self.internal_increase_protocol_fee(&token_a, &U128::from(0));
101    self.internal_increase_protocol_fee(&token_b, &U128::from(0));
102
103
```

```
104     self.internal_increase_locked_in_ref_asset(&token_a, &U128::from(0));
105     self.internal_increase_locked_in_ref_asset(&token_b, &U128::from(0));
106
107
108     self.internal_set_oracle(&token_a, token_a_oracle_id_op);
109     self.internal_set_oracle(&token_b, token_b_oracle_id_op);
110
111
112     self.internal_storage_deposit(&env::current_account_id(), &token_a,
113                                   REGISTER_TOKEN_STORAGE_FEE);
114     self.internal_storage_deposit(&env::current_account_id(), &token_b,
115                                   REGISTER_TOKEN_STORAGE_FEE);
116
117     self.internal_ref_storage_deposit(&env::current_account_id(), REGISTER_TOKEN_STORAGE_FEE);
118 }
```

Listing 2.2: dca\_owner.rs

```
9  pub fn create_dca(&mut self, name: String, token_in: AccountId, token_out: AccountId,
10                    single_amount_in: U128,
11                    start_time: u64, interval_time: u64, count: u16, lowest_price: u64,
12                    highest_price: u64, slippage: u16) -> bool {
13      // record storage fee
14      let initial_storage_usage = env::storage_usage();
15      let user = env::predecessor_account_id();
16      require!(slippage >= MIN_SLIPPAGE, SLIPPAGE_TOO_SMALL);
17      require!(start_time > env::block_timestamp_ms(), INVALID_START_TIME);
18      require!(self.deposit_limit_map.contains_key(&token_in) && self.deposit_limit_map.
19                contains_key(&token_out), INVALID_TOKEN);
20      if self.status != DCASStatus::Running {
21          self.internal_create_bot_refund_with_near(&user, &token_in, &token_out, env::
22            attached_deposit(), PAUSE_OR_SHUTDOWN);
23          return false;
24      }
25      let total_amount_in = single_amount_in.0 * (count as u128);
26      if self.internal_get_user_balance(&user, &token_in).0 < total_amount_in {
27          self.internal_create_bot_refund_with_near(&user, &token_in, &token_out, env::
28            attached_deposit(), LESS_TOKEN_IN);
29          return false;
30      }
31      // create id
32      let next_id = self.internal_get_and_use_next_id().to_string();
33      let next_dca_key = self.internal_get_dca_key(next_id);
34      let dca_vault = DCAVault {
35          name,
36          user: user.clone(),
37          id: next_dca_key.clone(),
38          closed: false,
39          token_in: token_in.clone(),
40          token_out,
41          start_time,
42          interval_time,
```

```
38     single_amount_in,
39     count,
40     execute_count: 0,
41     lowest_price,
42     highest_price,
43     left_amount_in: U128::from(total_amount_in),
44     buy_amount_record: U128::from(0),
45     slippage,
46     process: DCA_STATUS_NORMAL,
47     locked: false,
48     need_withdraw_amount: U128::from(0),
49     buy_amount_to_user: false,
50 };
51 self.dca_vault_map.insert(&next_dca_key, &dca_vault);
52 emit::create_dca(dca_vault);
53 // add locked asset
54 self.internal_transfer_assets_to_lock(&user, &token_in, U128::from(total_amount_in));
55
56
57 // refund storage fee
58 self.internal_refund_deposit(env::attached_deposit(), initial_storage_usage, &user);
59 return true;
60 }
```

**Listing 2.3:** dca.rs

```
121 pub fn token_storage_deposit(&mut self, user: AccountId, token: AccountId) {
122     require!(env::attached_deposit() == BASE_CREATE_STORAGE_FEE);
123     require!(self.deposit_limit_map.contains_key(&token), INVALID_TOKEN);
124     let initial_storage_usage = env::storage_usage();
125     self.internal_register_token_for_user(&user, &token);
126     self.internal_refund_deposit(BASE_CREATE_STORAGE_FEE, initial_storage_usage, &env::
        predecessor_account_id());
127 }
```

**Listing 2.4:** dca.rs

**Impact** The created [DCA](#) may not work.

**Suggestion** Add a check in the function `set_min_deposit()` to ensure that the `global_balances_map` includes the provided `token_id`.

### 2.1.2 Lack of refunding EXECUTE\_DCA\_FEE in function `execute_dca()`

**Severity** Low

**Status** Confirmed

**Introduced by** [Version 1](#)

**Description** According to the design, invoking the `execute_dca()` function requires paying a certain amount of [NEAR](#) (i.e., `EXECUTE_DCA_FEE`). However, the function performs multiple cross-contract calls, and the logic for handling failed results in callback functions does not refund this fee.

```
61  #[payable]
62  pub fn execute_dca(&mut self, vault_id: String, swap_msg: String) {
63      require!(env::attached_deposit() == EXECUTE_DCA_FEE);
64      require!(self.status == DCASStatus::Running, PAUSE_OR_SHUTDOWN);
65      require!(self.market_user_map.contains_key(&(env::predecessor_account_id())), INVALID_USER)
66      ;
67      require!(self.market_user_map.get(&(env::predecessor_account_id())).unwrap(), INVALID_USER)
68      ;
69      require!(self.dca_vault_map.contains_key(&vault_id), INVALID_VAULT_ID);
70      let mut dca_vault = self.dca_vault_map.get(&vault_id).unwrap();
71      require!(!dca_vault.locked, LOCKED);
72      require!(!dca_vault.closed, DCA_CLOSED);
73      self.internal_check_dca_buy_available(&dca_vault);
74      if dca_vault.process == DCA_STATUS_SWAPPED {
75          self.internal_ref_withdraw(&mut dca_vault);
76          return;
77      }
78      let pair_key = self.internal_get_pair_key(&dca_vault.token_in, &dca_vault.token_out);
79      let path_op = self.recorded_pair_path.get(&pair_key);
80      // execute buy
81      if self.internal_check_need_oracle(&dca_vault) {
82          // require oracle
83          self.get_price_for_execute(&mut dca_vault, swap_msg);
84      } else if path_op.is_some() && path_op.clone().unwrap().len() > 0 {
85          // check mint amount out
86          let single_amount_out = dca_vault.single_amount_in.0;
87          self.internal_ref_estimate(&mut dca_vault, swap_msg, path_op.unwrap(), 0,
88                                  single_amount_out);
89      } else {
90          // direct buy
91          self.internal_execute_buy(&mut dca_vault, swap_msg, None, None);
92      }
93  }
```

**Listing 2.5:** dca.rs

**Impact** Extra execution fees are charged.

**Suggestion** Refund fees when the whole process of interacting with [Ref-Exchange](#) is not completed.

**Feedback from the project** The current `EXECUTE_DCA_FEE` is only used to cover 1yocto. The fee is very low and there is no need to consider refunding it.

### 2.1.3 Lack of check on DCA status in function `close_dca()`

**Severity** High

**Status** Fixed in [Version 2](#)

**Introduced by** [Version 1](#)

**Description** The function `close_dca()` closes the [DCA](#) vault, but it does not check whether the [DCA](#) vault is in a `DCA_STATUS_NORMAL` state or locked.

```

92  #[payable]
93  pub fn close_dca(&mut self, vault_id: String) {
94      assert_one_yocto();
95      require!(self.dca_vault_map.contains_key(&vault_id), VAULT_NOT_EXIST);
96      let mut dca_vault = self.dca_vault_map.get(&vault_id).unwrap();
97      require!(!dca_vault.closed, INVALID_BOT_STATUS);
98      // check permission, user self close
99      require!(env::predecessor_account_id() == dca_vault.user, INVALID_USER);
100
101
102      dca_vault.closed = true;
103      self.internal_transfer_assets_to_unlock(&(dca_vault.user), &(dca_vault.token_in), dca_vault
        .left_amount_in.clone());
104      self.internal_transfer_assets_to_unlock(&(dca_vault.user), &(dca_vault.token_out),
        dca_vault.buy_amount_record.clone());
105      // update dca_vault info
106      self.dca_vault_map.insert(&vault_id, &dca_vault);
107      // withdraw
108      self.internal_withdraw_all(&(dca_vault.user), &(dca_vault.token_in));
109      self.internal_withdraw_all(&(dca_vault.user), &(dca_vault.token_out));
110
111
112      emit::close_dca(&env::predecessor_account_id(), dca_vault.id.clone(), dca_vault.
        left_amount_in.0, dca_vault.buy_amount_record.0);
113  }

```

**Listing 2.6:** dca.rs

**Impact** DCA funds in a non-normal state will be stuck in the [Ref-Exchange](#).

**Suggestion** Add check in function `close_dca()`.

### 2.1.4 Lack of check on contract status in function `token_storage_deposit()`

**Severity** Low

**Status** Fixed in [Version 2](#)

**Introduced by** [Version 1](#)

**Description** In the `dca.rs` file, users can deposit a storage fee through the function `token_storage_deposit()`, but this function does not check if the contract is running. Specifically, if the contract is in a paused state, depositing a storage fee at this time is meaningless.

```

121  pub fn token_storage_deposit(&mut self, user: AccountId, token: AccountId) {
122      require!(env::attached_deposit() == BASE_CREATE_STORAGE_FEE);
123      require!(self.deposit_limit_map.contains_key(&token), INVALID_TOKEN);
124      let initial_storage_usage = env::storage_usage();
125      self.internal_register_token_for_user(&user, &token);
126      self.internal_refund_deposit(BASE_CREATE_STORAGE_FEE, initial_storage_usage, &env::
        predecessor_account_id());
127  }

```

**Listing 2.7:** dca\_owner.rs

**Impact** When the contract is in a paused state, depositing a storage fee by users is meaningless.

**Suggestion** Add a check to ensure that the contract is running.

### 2.1.5 Incorrect check of promise results in function `callback_do_withdraw()`

**Severity** High

**Status** Fixed in [Version 2](#)

**Introduced by** [Version 1](#)

**Description** The callback function `callback_do_withdraw()` is designed to handle the promise result returned by cross-contract invocation `withdraw()` sent to `Ref-Exchange`. Meanwhile, in `Ref-Exchange`, `withdraw()` invokes function `ft_transfer()` to send the tokens. When the function `ft_transfer()` fails, function `exchange_callback_post_withdraw()` will help to recover the state and the `is_promise_success()` would return true in function `callback_do_withdraw()`. In this case, the function would consider the contract has withdrawn the tokens successfully, which is incorrect.

```

75  #[private]
76  pub fn callback_do_withdraw(&mut self, dca_vault: &mut DCAVault, amount_in: u128, amount_out:
    u128) {
77      if !is_promise_success() {
78          emit::ref_withdraw_failed(&dca_vault.user, amount_out, &dca_vault.token_out);
79          self.internal_unlock_dca_vault(dca_vault);
80          return;
81      }
82      emit::ref_withdraw_succeeded(&dca_vault.user, amount_out, &dca_vault.token_out);
83      // complete once dca
84      // calculate protocol fee
85      let (real_amount_out, protocol_fee) = self.internal_calculate_protocol_fee(amount_out);
86
87
88      self.internal_record_ref_normal(dca_vault, amount_in, real_amount_out);
89      // update asset
90      self.internal_increase_locked_assets(&dca_vault.user, &dca_vault.token_out, &U128::from(
        real_amount_out));
91      // update global asset
92      self.internal_increase_global_asset(&dca_vault.token_out, &U128::from(amount_out));
93      // add protocol asset
94      self.internal_increase_protocol_fee(&dca_vault.token_out, &U128::from(protocol_fee));
95  }

```

**Listing 2.8:** `dca_callback.rs`

```

289  #[private]
290  pub fn exchange_callback_post_withdraw(
291      &mut self,
292      token_id: AccountId,
293      sender_id: AccountId,
294      amount: U128,
295  ) -> U128 {

```

```
296     assert_eq!(
297         env::promise_results_count(),
298         1,
299         "{}",
300         ERR25_CALLBACK_POST_WITHDRAW_INVALID
301     );
302     match env::promise_result(0) {
303         PromiseResult::NotReady => unreachable!(),
304         PromiseResult::Successful(_) => amount,
305         PromiseResult::Failed => {
306             // This reverts the changes from withdraw function.
307             // If account doesn't exist, deposits to the owner's account as lostfound.
308             let mut failed = false;
309             if let Some(mut account) = self.internal_get_account(&sender_id) {
310                 if account.deposit_with_storage_check(&token_id, amount.0) {
311                     // cause storage already checked, here can directly save
312                     self.accounts.insert(&sender_id, &account.into());
313                 } else {
314                     // we can ensure that internal_get_account here would NOT cause a version
315                     // upgrade,
316                     // cause it is callback, the account must be the current version or non-
317                     // exist,
318                     // so, here we can just leave it without insert, won't cause storage
319                     // collection inconsistency.
320                     env::log(
321                         format!(
322                             "Account {} has not enough storage. Depositing to owner.",
323                             sender_id
324                         ),
325                         .as_bytes(),
326                     );
327                     failed = true;
328                 }
329             } else {
330                 env::log(
331                     format!(
332                         "Account {} is not registered. Depositing to owner.",
333                         sender_id
334                     ),
335                     .as_bytes(),
336                 );
337                 failed = true;
338             }
339             if failed {
340                 self.internal_lostfound(&token_id, amount.0);
341             }
342         }
343     }
```

**Listing 2.9:** ref-contracts/ref-exchange/account\_deposit.rs

**Impact** Due to incorrect handling of the promise result, the contract state would be incorrectly updated.

**Suggestion** Parse the promise result, and if the result is 0, treat the operation withdrawal as failed and handle it accordingly.

### 2.1.6 Inconsistent update time between `global_balances_map` and token balance

**Severity** Medium

**Status** Fixed in [Version 2](#)

**Introduced by** [Version 1](#)

**Description** The function `callback_do_withdraw()` updates the `global_balances_map` to synchronize token balance changes. However, due to asynchronous invocation, the token balance can be updated earlier than the `global_balances_map`. In this case, the owner can invoke the function `withdraw_unowned_asset()` to withdraw assets belonging to the user.

```
75  #[private]
76  pub fn callback_do_withdraw(&mut self, dca_vault: &mut DCAVault, amount_in: u128, amount_out:
    u128) {
77      if !is_promise_success() {
78          emit::ref_withdraw_failed(&dca_vault.user, amount_out, &dca_vault.token_out);
79          self.internal_unlock_dca_vault(dca_vault);
80          return;
81      }
82      emit::ref_withdraw_succeeded(&dca_vault.user, amount_out, &dca_vault.token_out);
83      // complete once dca
84      // calculate protocol fee
85      let (real_amount_out, protocol_fee) = self.internal_calculate_protocol_fee(amount_out);
86
87
88      self.internal_record_ref_normal(dca_vault, amount_in, real_amount_out);
89      // update asset
90      self.internal_increase_locked_assets(&dca_vault.user, &dca_vault.token_out, &U128::from(
        real_amount_out));
91      // update global asset
92      self.internal_increase_global_asset(&dca_vault.token_out, &U128::from(amount_out));
93      // add protocol asset
94      self.internal_increase_protocol_fee(&dca_vault.token_out, &U128::from(protocol_fee));
95  }
```

**Listing 2.10:** `dca_callback.rs`

**Impact** Owner can withdraw user assets.

**Suggestion** Revise the logic accordingly.

### 2.1.7 Incorrect logic in function `withdraw_asset_from_ref()`

**Severity** Medium

**Status** Fixed in [Version 2](#)



## Introduced by [Version 1](#)

**Description** In the `dca_owner.rs` file, the `owner` can withdraw assets that users have deposited in the `Ref-Exchange` via the function `withdraw_asset_from_ref()`. According to the implementation of the function `withdraw()` in the `Ref-Exchange`, an input amount of zero signifies the withdrawal of all assets. If the owner's private key is lost or maliciously used, it could result in losses for the users.

```
45 pub fn withdraw_asset_from_ref(&mut self, token: AccountId, amount: U128) {
46     self.assert_owner();
47     require!(self.internal_get_locked_in_ref_asset(&token) >= amount.0, INVALID_TOKEN);
48     self.internal_ref_withdraw_directly(&env::current_account_id(), &token, &amount);
49 }
```

**Listing 2.11:** `dca_owner.rs`

```
87 pub fn internal_ref_withdraw_directly(&mut self, user: &AccountId, token: &AccountId, amount:
    &U128) {
88     ext_ref::ext(self.ref_exchange_id.clone())
89         .with_attached_deposit(1)
90         .with_static_gas(GAS_FOR_WITHDRAW)
91         .withdraw(
92             token.clone(),
93             amount.clone(),
94             None,
95             None
96         ).then(
97             Self::ext(env::current_account_id())
98                 .with_static_gas(GAS_FOR_WITHDRAW_CALL_BACK)
99                 .callback_do_withdraw_directly(user, token, amount)
100         );
101 }
```

**Listing 2.12:** `dca_private.rs`

```
353 pub fn withdraw(
354     &mut self,
355     token_id: ValidAccountId,
356     amount: U128,
357     unregister: Option<bool>,
358     skip_unwrap_near: Option<bool>
359 ) -> Promise {
360     assert_one_yocto();
361     self.assert_contract_running();
362     let token_id: AccountId = token_id.into();
363     // feature frozenlist
364     self.assert_no_frozen_tokens(&[token_id.clone()]);
365     let sender_id = env::predecessor_account_id();
366     let mut account = self.internal_unwrap_account(&sender_id);
367
368     // get full amount if amount param is 0
369     let mut amount: u128 = amount.into();
370     if amount == 0 {
371         amount = account.get_balance(&token_id).expect(ERR21_TOKEN_NOT_REG);
```

```

372     }
373     assert!(amount > 0, "{}", ERR29_ILLEGAL_WITHDRAW_AMOUNT);
374
375     // Note: subtraction and deregistration will be reverted if the promise fails.
376     account.withdraw(&token_id, amount);
377     if unregister == Some(true) {
378         account.unregister(&token_id);
379     }
380     self.internal_save_account(&sender_id, account);
381     self.internal_send_tokens(&sender_id, &token_id, amount, skip_unwrap_near)
382 }

```

**Listing 2.13:** ref-contracts/ref-exchange/account\_deposit.rs

**Impact** If the `owner`'s private key is lost or maliciously exploited, users' assets could be at risk of loss.

**Suggestion** Revise the logic accordingly.

## 2.1.8 Incorrect check in function `internal_execute_buy()`

**Severity** Medium

**Status** Fixed in [Version 2](#)

**Introduced by** [Version 1](#)

**Description** The check of the user's locked balance inside the internal function `internal_execute_buy()` is incorrect. Specifically, completing a single buy process requires going through three cross-contract invocations: deposit, swap, and withdraw. Once the deposit succeeds, in the callback function `callback_do_deposit()`, the user's corresponding locked balance of that token will decrease. However, if the swap fails, the corresponding locked balance would not change. In this case, if the locked balance is 0 after the deposit, when the market user tries to invoke `execute_dca()` to complete the failed swap, the check in `internal_execute_buy()` would result in revert, which is incorrect. Besides, this check does not actually take into account that a user can have multiple vaults containing the same token. The check of locked balances in this scenario is not reasonable.

```

11 pub fn internal_execute_buy(&mut self, dca_vault: &mut DCAVault, swap_msg: String,
12     price_list_op: Option<Vec<Price>>, estimate_amount_out_op: Option<u128>) -> bool {
13     require!(self.status == DCASStatus::Running, PAUSE_OR_SHUTDOWN);
14     require!(!dca_vault.locked, LOCKED);
15     require!(!dca_vault.closed, DCA_CLOSED);
16     require!(self.internal_get_user_locked_balance(&dca_vault.user, &dca_vault.token_in).0 >=
17         dca_vault.single_amount_in.0, LESS_TOKEN_IN);
18     let swap_info = serde_json::from_str::<SwapMessage>(&swap_msg).expect(INVALID_EXECUTE_MSG);
19     // check swap param
20     let (amount_in, amount_out) = self.internal_check_swap_info(&swap_info, &dca_vault);
21     // check oracle
22     self.internal_check_oracle_price(&dca_vault, price_list_op, amount_in, amount_out);
23     // check estimate amount out
24     self.internal_check_estimate_amount_out(amount_out, estimate_amount_out_op, dca_vault.slippage);

```

```
23
24
25     self.internal_lock_dca_vault(dca_vault);
26     if dca_vault.process == DCA_STATUS_NORMAL {
27         // 1.deposit,2.swap,3.withdraw
28         self.internal_ref_deposit(dca_vault, amount_in, &swap_info);
29     } else if dca_vault.process == DCA_STATUS_DEPOSITED {
30         // 1.swap, 2.withdraw
31         self.internal_ref_swap(dca_vault, amount_in, &swap_info);
32     }
33     // else {
34     //     // 1.withdraw
35     //     self.internal_ref_withdraw(dca_vault);
36     // }
37     return true;
38 }
```

Listing 2.14: dca\_private.rs

```
11  #[private]
12  pub fn callback_do_deposit(&mut self, dca_vault: &mut DCAVault, amount_in: u128, swap_info: &
    SwapMessage) {
13      if !is_promise_success() {
14          // deposit error, assets still on DCA contract, so don't need to do anything
15          emit::ref_deposit_failed(&dca_vault.user, amount_in, &dca_vault.token_in);
16          self.internal_unlock_dca_vault(dca_vault);
17          return;
18      }
19      let cross_call_result = promise_result_as_success().expect(ERR102_CROSS_CONTRACT_FAILED);
20      let amount_in_real = serde_json::from_slice::<U128>(&cross_call_result).unwrap().0;
21      if amount_in != amount_in_real {
22          // deposit must same as the total
23          emit::ref_deposit_failed(&dca_vault.user, amount_in, &dca_vault.token_in);
24          self.internal_unlock_dca_vault(dca_vault);
25          if amount_in_real == 0 {
26              // nothing to do, deposit 0, so don't need to do anything
27              return;
28          }
29          // need owner to withdraw from ref
30          emit::need_owner_withdraw_from_ref(&dca_vault.user, amount_in_real, &dca_vault.token_in
            );
31          // add ref locked asset
32          self.internal_increase_locked_in_ref_asset(&dca_vault.token_in, &U128::from(
            amount_in_real));
33          // reduce asset from global
34          // self.internal_reduce_locked_assets(&dca_vault.user, &dca_vault.token_in, &U128::from
            (amount_in));
35          self.internal_reduce_global_asset(&dca_vault.token_in, &U128::from(amount_in_real));
36          // withdraw
37          self.internal_ref_withdraw_directly(&dca_vault.user, &dca_vault.token_in, &U128::from(
            amount_in_real));
38          return;
39      }
```

```
40     emit::ref_deposit_success(&dca_vault.user, amount_in, &dca_vault.token_in);
41     // reduce asset from user locked asset and global
42     self.internal_reduce_locked_assets(&dca_vault.user, &dca_vault.token_in, &U128::from(
43         amount_in));
44     self.internal_reduce_global_asset(&dca_vault.token_in, &U128::from(amount_in_real));
45     // execute deposit record
46     self.internal_record_ref_deposited(dca_vault);
47     // execute swap
48     self.internal_ref_swap(dca_vault, amount_in, swap_info);
49 }
```

**Listing 2.15:** dca\_callback.rs

**Impact** If swap fails, the market user cannot swap again by invoking the function `execute_dca()` due to the incorrect check inside the function `internal_execute_buy()`.

**Suggestion** Remove the check.

### 2.1.9 Potential user losses due to incorrect swap path

**Severity** High

**Status** Fixed in [Version 2](#)

**Introduced by** [Version 1](#)

**Description** The function `internal_check_swap_info()` allows swap paths with multiple duplicate pools, such as `NEAR-USDC`, `NEAR-USDC`. However, in the function `callback_do_swap()`, only the swap result of the last pool is used as the `need_withdraw_amount`. The swap result tokens from the previous pools will remain in the `Ref-Exchange`.

```
65 pub fn internal_check_swap_info(&self, swap_info: &SwapMessage, dca_vault: &DCAVault) -> (u128
66     , u128) {
67     let mut amount_in = 0;
68     let mut amount_out = 0;
69     require!(!swap_info.actions.is_empty(), INVALID_SWAP_ACTIONS);
70
71     let mut pre_action: Option<&Action> = None;
72     for (index, action) in swap_info.actions.iter().enumerate() {
73         if index == 0 {
74             require!(action.token_in == dca_vault.token_in, INVALID_TOKEN_IN);
75         }
76         if index == swap_info.actions.len() - 1 {
77             require!(action.token_out == dca_vault.token_out, INVALID_TOKEN_OUT);
78         }
79         if action.token_in == dca_vault.token_in {
80             amount_in += action.amount_in.expect(ERR100_WRONG_MSG_FORMAT).0
81         }
82         if action.token_out == dca_vault.token_out {
83             amount_out += action.min_amount_out.0;
84         }
85         if pre_action.clone().is_some() && action.token_in != dca_vault.token_in {
86             // must be a chain
```

```
87         require!(pre_action.clone().unwrap().token_out == action.token_in,
88                 INVALID_SWAP_CHAIN_PRE_ACTION_OUT_MUST_ACTION_IN);
89         require!(action.amount_in.is_none(), INVALID_AMOUNT_IN);
90     }
91     if pre_action.clone().is_some() && action.token_in == dca_vault.token_in {
92         require!(pre_action.clone().unwrap().token_out == dca_vault.token_out,
93                 INVALID_SWAP_CHAIN_PRE_ACTION_OUT_MUST_ACTION_OUT);
94         require!(action.amount_in.is_some(), INVALID_AMOUNT_IN);
95     }
96     pre_action = Some(action);
97 }
98 require!(amount_in == dca_vault.single_amount_in.0, INVALID_AMOUNT_IN);
99 return (amount_in, amount_out);
100 }
```

**Listing 2.16:** dca\_check.rs

```
50 #[private]
51 pub fn callback_do_swap(&mut self, dca_vault: &mut DCAVault, amount_in: u128) {
52     if !is_promise_success() {
53         emit::ref_swap_failed(&dca_vault.user, amount_in, &dca_vault.token_in, &dca_vault.
54             token_out);
55         self.internal_unlock_dca_vault(dca_vault);
56         return;
57     }
58
59     let cross_call_result = promise_result_as_success().expect(ERR102_CROSS_CONTRACT_FAILED);
60     let action_result = serde_json::from_slice::<ActionResult>(&cross_call_result).unwrap();
61     match action_result {
62         ActionResult::None => {
63             emit::ref_swap_failed(&dca_vault.user, amount_in, &dca_vault.token_in, &dca_vault.
64                 token_out);
65             self.internal_unlock_dca_vault(dca_vault);
66         }
67         ActionResult::Amount(amount) => {
68             emit::ref_swap_success(&dca_vault.user, amount_in, amount.0, &dca_vault.token_in, &
69                 dca_vault.token_out);
70             // record to swapped
71             self.internal_record_ref_swapped(dca_vault, amount.clone());
72             // withdraw
73             self.internal_ref_withdraw(dca_vault);
74         }
75     }
76 }
```

**Listing 2.17:** dca\_callback.rs

**Impact** User funds may be stuck in [Ref-Exchange](#).

**Suggestion** Do not support swap paths with duplicate pools.

## 2.1.10 Unscaled expo values in Pyth-Oracle integration

**Severity** High

**Status** Fixed in [Version 2](#)

**Introduced by** [Version 1](#)

**Description** Some logic within the protocol relies on the [Pyth-Oracle](#). The contract invokes [Pyth-Oracle](#)'s function `get_price()` to obtain the price of a specific token, which returns both the `price` and the `expo`. However, the protocol does not scale the `price` returned by the [Pyth-Oracle](#) according to the `expo`. It is assumed that the `expo` values for all tokens are consistent. Otherwise, the calculated price can be wrong.

```

20  pub fn internal_check_oracle_price(&self, dca_vault: &DCAVault, price_list_op: Option<Vec<
    Price>>, amount_in: u128, amount_out: u128) {
21      let in_meta_decimal = self.token_decimal_map.get(&dca_vault.token_in).unwrap();
22      let out_meta_decimal = self.token_decimal_map.get(&dca_vault.token_out).unwrap();
23      let swap_price = (BigDecimal::from(amount_out * (10 as u128).pow(in_meta_decimal as u32)) *
        BigDecimal::from(PRICE_DENOMINATOR) / BigDecimal::from(amount_in * (10 as u128).pow(
        out_meta_decimal as u32))).round_down_u128();
24      self.internal_check_price_limit(swap_price, &dca_vault);
25      if !self.internal_check_need_oracle(&dca_vault) {
26          return;
27      }
28      let price_list = price_list_op.unwrap();
29      let in_price = &price_list[0];
30      let out_price = &price_list[1];
31      require!(in_price.publish_time as u64 * 1000 + self.oracle_valid_time.clone() >= env::
        block_timestamp_ms(), PRICE_EXPIRED);
32      require!(out_price.publish_time as u64 * 1000 + self.oracle_valid_time.clone() >= env::
        block_timestamp_ms(), PRICE_EXPIRED);
33      let in_scaled_price = BigDecimal::from(in_price.price.0 as u64) * BigDecimal::from(
        PRICE_SCALED_DENOMINATOR) / BigDecimal::from(in_price.expo.abs() as u64);
34      let out_scaled_price = BigDecimal::from(out_price.price.0 as u64) * BigDecimal::from(
        PRICE_SCALED_DENOMINATOR) / BigDecimal::from(out_price.expo.abs() as u64);
35
36      let oracle_price = (in_scaled_price * BigDecimal::from(PRICE_DENOMINATOR) /
        out_scaled_price).round_down_u128();
37      self.internal_check_price_limit(oracle_price, &dca_vault);
38
39      if swap_price >= oracle_price {
40          require!((swap_price - oracle_price) * SLIPPAGE_DENOMINATOR as u128 / swap_price <=
        dca_vault.slippage as u128, OUT_OF_SLIPPAGE);
41      } else {
42          require!((oracle_price - swap_price) * SLIPPAGE_DENOMINATOR as u128 / swap_price <=
        dca_vault.slippage as u128, OUT_OF_SLIPPAGE);
43      }
44  }

```

**Listing 2.18:** dca\_check.rs

**Impact** The prices of `token_in` and `token_out` may be calculated incorrectly.

**Suggestion** Revise the logic to normalize the `price` based on the return values from `Pyth-Oracle`.

### 2.1.11 Potential user losses due to manipulated `amount_out`

**Severity** High

**Status** Fixed in [Version 2](#)

**Introduced by** [Version 1](#)

**Description** The function `internal_check_swap_info()` is used to check if the swap path is valid and accumulate the `amount_in` and `amount_out` to return to the corresponding function to calculate the swap price. The swap price has to meet the vault's defined price limit before swapping. However, the check in `internal_check_swap_info()` is not sufficient. As long as the action's `token_out` matches the vault's `token_out` in the path, it will accumulate the `amount_out`, allowing the swap price to be manipulated to the target value by constructing the path. Specifically, assuming the `tokenIn` is A, and `tokenOut` is C, the path could be A->B, B->C, C->B, B->C, C->B...B->C, looping in the middle to repeatedly accumulate the `amount_out`, ultimately making the swap price reach the target value.

```
65 pub fn internal_check_swap_info(&self, swap_info: &SwapMessage, dca_vault: &DCAVault) -> (u128
    , u128) {
66     let mut amount_in = 0;
67     let mut amount_out = 0;
68     require!(!swap_info.actions.is_empty(), INVALID_SWAP_ACTIONS);
69
70
71     let mut pre_action: Option<&Action> = None;
72     for (index, action) in swap_info.actions.iter().enumerate() {
73         if index == 0 {
74             require!(action.token_in == dca_vault.token_in, INVALID_TOKEN_IN);
75         }
76         if index == swap_info.actions.len() - 1 {
77             require!(action.token_out == dca_vault.token_out, INVALID_TOKEN_OUT);
78         }
79         if action.token_in == dca_vault.token_in {
80             amount_in += action.amount_in.expect(ERR100_WRONG_MSG_FORMAT).0
81         }
82         if action.token_out == dca_vault.token_out {
83             amount_out += action.min_amount_out.0;
84         }
85         if pre_action.clone().is_some() && action.token_in != dca_vault.token_in {
86             // must be a chain
87             require!(pre_action.clone().unwrap().token_out == action.token_in,
                INVALID_SWAP_CHAIN_PRE_ACTION_OUT_MUST_ACTION_IN);
88             require!(action.amount_in.is_none(), INVALID_AMOUNT_IN);
89         }
90         if pre_action.clone().is_some() && action.token_in == dca_vault.token_in {
91             require!(pre_action.clone().unwrap().token_out == dca_vault.token_out,
                INVALID_SWAP_CHAIN_PRE_ACTION_OUT_MUST_ACTION_OUT);
92             require!(action.amount_in.is_some(), INVALID_AMOUNT_IN);
93         }
94     }
```

```

94     pre_action = Some(action);
95 }
96 require!(amount_in == dca_vault.single_amount_in.0, INVALID_AMOUNT_IN);
97 return (amount_in, amount_out);
98 }

```

**Listing 2.19:** dca\_check.rs

**Impact** The swap price can be manipulated by constructing an invalid swap path.

**Suggestion** Implement corresponding check logic accordingly.

### 2.1.12 Lack of setting static gas in function `after_withdraw_near()`

**Severity** Low

**Status** Fixed in [Version 2](#)

**Introduced by** [Version 1](#)

**Description** The function `withdraw_near()` does not attach static gas to the function `after_withdraw_near()`. In this case, the callback function may fail due to insufficient gas, resulting in incorrect contract state.

```

15 pub fn withdraw_near(&mut self, user: &AccountId, amount: u128) {
16     ext_wnear::ext(self.wnear.clone())
17     .with_attached_deposit(ONE_YOCTO)
18     .near_withdraw(U128::from(amount))
19     .then(
20         Self::ext(env::current_account_id())
21         .after_withdraw_near(
22             user,
23             amount,
24         )
25     );
26 }

```

**Listing 2.20:** wnear.rs

```

38 #[private]
39 fn after_withdraw_near(&mut self, user: &AccountId, amount: u128) -> bool {
40     let promise_success = is_promise_success();
41     if !promise_success.clone() {
42         emit::wrap_near_error(user, 0, amount, false);
43         self.internal_increase_asset(user, &self.wnear.clone(), &(U128::from(amount)));
44     } else {
45         self.internal_ft_transfer_near(user, amount, true);
46     }
47     promise_success
48 }

```

**Listing 2.21:** wnear.rs

```

102 pub fn internal_ft_transfer_near(&mut self, receiver_id: &AccountId, amount: Balance,
    effect_global_balance: bool) -> Promise {

```



```

103     require!(self.internal_get_remaining_gas() >= GAS_FOR_FT_TRANSFER, LESS_GAS);
104     if effect_global_balance {
105         // reduce from global asset
106         self.internal_reduce_global_asset(&self.wnear.clone(), &U128::from(amount))
107     }
108     Promise::new(receiver_id.clone()).transfer(amount)
109 }

```

**Listing 2.22:** token.rs

**Impact** The `global_balances_map` will not update correctly due to insufficient gas.

**Suggestion** Attach enough static gas to function `after_withdraw_near()`.

### 2.1.13 Lack of depositing storage fee for `token_out`

**Severity** Medium

**Status** Fixed in [Version 2](#)

**Introduced by** [Version 1](#)

**Description** Users can create a DCA through the function `create_dca()`, provided they have already deposited the necessary storage fee. Within `create_dca()`, the function `internal_transfer_assets_to_lock()` transfers the user's unlocked balance to their locked balance and ensures that a storage fee has been paid for `token_in` by the user. However, there is no requirement for users to deposit a storage fee for `token_out`. Specifically, when users complete a periodic investment, they utilize the function `internal_ref_withdraw()` to withdraw assets from the Ref-Exchange. In the callback function `callback_do_withdraw()`, the function `internal_increase_locked_assets()` directly records the quantity of `token_out` exchanged into the user's locked balance without requesting the storage fee, which is incorrect.

```

62 pub fn create_dca(&mut self, name: String, token_in: AccountId, token_out: AccountId,
63     single_amount_in: U128,
64     start_time: u64, interval_time: u64, count: u16, lowest_price: u64,
65     highest_price: u64, slippage: u16) -> bool {
66     // record storage fee
67     let initial_storage_usage = env::storage_usage();
68     let user = env::predecessor_account_id();
69     require!(slippage >= MIN_SLIPPAGE, SLIPPAGE_TOO_SMALL);
70     require!(start_time > env::block_timestamp_ms(), INVALID_START_TIME);
71     require!(self.deposit_limit_map.contains_key(&token_in) && self.deposit_limit_map.
72         contains_key(&token_out), INVALID_TOKEN);
73     if self.status != DCASStatus::Running {
74         self.internal_create_bot_refund_with_near(&user, &token_in, &token_out, env::
75             attached_deposit(), PAUSE_OR_SHUTDOWN);
76         return false;
77     }
78     let total_amount_in = single_amount_in.0 * (count as u128);
79     if self.internal_get_user_balance(&user, &token_in).0 < total_amount_in {
80         self.internal_create_bot_refund_with_near(&user, &token_in, &token_out, env::
81             attached_deposit(), LESS_TOKEN_IN);
82         return false;
83     }
84 }

```

```

78     }
79     // create id
80     let next_id = self.internal_get_and_use_next_id().to_string();
81     let next_dca_key = self.internal_get_dca_key(next_id);
82     let dca_vault = DCAVault {
83         name,
84         user: user.clone(),
85         id: next_dca_key.clone(),
86         closed: false,
87         token_in: token_in.clone(),
88         token_out,
89         start_time,
90         interval_time,
91         single_amount_in,
92         count,
93         execute_count: 0,
94         lowest_price,
95         highest_price,
96         left_amount_in: U128::from(total_amount_in),
97         buy_amount_record: U128::from(0),
98         slippage,
99         process: DCA_STATUS_NORMAL,
100        locked: false,
101        need_withdraw_amount: U128::from(0),
102        buy_amount_to_user: false,
103    };
104    self.dca_vault_map.insert(&next_dca_key, &dca_vault);
105    emit::create_dca(dca_vault);
106    // add locked asset
107    self.internal_transfer_assets_to_lock(&user, &token_in, U128::from(total_amount_in));
108
109
110    // refund storage fee
111    self.internal_refund_deposit(env::attached_deposit(), initial_storage_usage, &user);
112    return true;
113 }

```

**Listing 2.23:** dca.rs

```

62 pub fn execute_dca(&mut self, vault_id: String, swap_msg: String) {
63     require!(env::attached_deposit() == EXECUTE_DCA_FEE);
64     require!(self.status == DCAStatus::Running, PAUSE_OR_SHUTDOWN);
65     require!(self.market_user_map.contains_key(&(env::predecessor_account_id())), INVALID_USER)
66     ;
67     require!(self.market_user_map.get(&(env::predecessor_account_id())).unwrap(), INVALID_USER)
68     ;
69     require!(self.dca_vault_map.contains_key(&vault_id), INVALID_VAULT_ID);
70     let mut dca_vault = self.dca_vault_map.get(&vault_id).unwrap();
71     require!(!dca_vault.locked, LOCKED);
72     require!(!dca_vault.closed, DCA_CLOSED);
73     self.internal_check_dca_buy_available(&dca_vault);
74     if dca_vault.process == DCA_STATUS_SWAPPED {
75         self.internal_ref_withdraw(&mut dca_vault);

```

```
74     return;
75 }
76 let pair_key = self.internal_get_pair_key(&dca_vault.token_in, &dca_vault.token_out);
77 let path_op = self.recorded_pair_path.get(&pair_key);
78 // execute buy
79 if self.internal_check_need_oracle(&dca_vault) {
80     // require oracle
81     self.get_price_for_execute(&mut dca_vault, swap_msg);
82 } else if path_op.is_some() && path_op.clone().unwrap().len() > 0 {
83     // check mint amount out
84     let single_amount_out = dca_vault.single_amount_in.0;
85     self.internal_ref_estimate(&mut dca_vault, swap_msg, path_op.unwrap(), 0,
86         single_amount_out);
87 } else {
88     // direct buy
89     self.internal_execute_buy(&mut dca_vault, swap_msg, None, None);
90 }
```

Listing 2.24: dca.rs

```
11 pub fn internal_execute_buy(&mut self, dca_vault: &mut DCAVault, swap_msg: String,
12     price_list_op: Option<Vec<Price>>, estimate_amount_out_op: Option<u128>) -> bool {
13     require!(self.status == DCAStatus::Running, PAUSE_OR_SHUTDOWN);
14     require!(!dca_vault.locked, LOCKED);
15     require!(!dca_vault.closed, DCA_CLOSED);
16     require!(self.internal_get_user_locked_balance(&dca_vault.user, &dca_vault.token_in).0 >=
17         dca_vault.single_amount_in.0, LESS_TOKEN_IN);
18     let swap_info = serde_json::from_str::<SwapMessage>(&swap_msg).expect(INVALID_EXECUTE_MSG);
19     // check swap param
20     let (amount_in, amount_out) = self.internal_check_swap_info(&swap_info, &dca_vault);
21     // check oracle
22     self.internal_check_oracle_price(&dca_vault, price_list_op, amount_in, amount_out);
23     // check estimate amount out
24     self.internal_check_estimate_amount_out(amount_out, estimate_amount_out_op, dca_vault.slippage);
25
26     self.internal_lock_dca_vault(dca_vault);
27     if dca_vault.process == DCA_STATUS_NORMAL {
28         // 1.deposit,2.swap,3.withdraw
29         self.internal_ref_deposit(dca_vault, amount_in, &swap_info);
30     } else if dca_vault.process == DCA_STATUS_DEPOSITED {
31         // 1.swap, 2.withdraw
32         self.internal_ref_swap(dca_vault, amount_in, &swap_info);
33     }
34     // else {
35     //     // 1.withdraw
36     //     self.internal_ref_withdraw(dca_vault);
37     // }
38     return true;
39 }
```

Listing 2.25: dca\_private.rs

```
55 pub fn internal_ref_swap(&mut self, dca_vault: &mut DCAVault, amount_in: u128, swap_info: &
    SwapMessage) {
56     // get referral
57     let referral = if swap_info.referral_id.is_some() { Some(AccountId::new_unchecked(swap_info
        .referral_id.clone().unwrap().to_string())) } else { None };
58     ext_ref::ext(self.ref_exchange_id.clone())
59         .with_attached_deposit(1)
60         .with_static_gas(GAS_FOR_SWAP)
61         .execute_actions(
62             swap_info.actions.clone(),
63             referral
64         ).then(
65             Self::ext(env::current_account_id())
66                 .with_static_gas(GAS_FOR_SWAP_CALL_BACK)
67                 .callback_do_swap(dca_vault, amount_in)
68         );
69 }
```

Listing 2.26: dca\_private.rs

```
51 pub fn callback_do_swap(&mut self, dca_vault: &mut DCAVault, amount_in: u128) {
52     if !is_promise_success() {
53         emit::ref_swap_failed(&dca_vault.user, amount_in, &dca_vault.token_in, &dca_vault.
            token_out);
54         self.internal_unlock_dca_vault(dca_vault);
55         return;
56     }
57
58
59     let cross_call_result = promise_result_as_success().expect(ERR102_CROSS_CONTRACT_FAILED);
60     let action_result = serde_json::from_slice::<ActionResult>(&cross_call_result).unwrap();
61     match action_result {
62         ActionResult::None => {
63             emit::ref_swap_failed(&dca_vault.user, amount_in, &dca_vault.token_in, &dca_vault.
                token_out);
64             self.internal_unlock_dca_vault(dca_vault);
65         }
66         ActionResult::Amount(amount) => {
67             emit::ref_swap_success(&dca_vault.user, amount_in, amount.0, &dca_vault.token_in, &
                dca_vault.token_out);
68             // record to swapped
69             self.internal_record_ref_swapped(dca_vault, amount.clone());
70             // withdraw
71             self.internal_ref_withdraw(dca_vault);
72         }
73     }
74 }
```

Listing 2.27: dca\_callback.rs

```
71 pub fn internal_ref_withdraw(&mut self, dca_vault: &mut DCAVault) {
72     ext_ref::ext(self.ref_exchange_id.clone())
73         .with_attached_deposit(1)
```

```

74         .with_static_gas(GAS_FOR_WITHDRAW)
75         .withdraw(
76             dca_vault.token_out.clone(),
77             dca_vault.need_withdraw_amount.clone(),
78             None,
79             None
80         ).then(
81             Self::ext(env::current_account_id())
82                 .with_static_gas(GAS_FOR_WITHDRAW_CALL_BACK)
83                 .callback_do_withdraw(dca_vault, dca_vault.single_amount_in.0, dca_vault.
                        need_withdraw_amount.0)
84         );
85     }

```

**Listing 2.28:** dca\_private.rs

```

76     pub fn callback_do_withdraw(&mut self, dca_vault: &mut DCAVault, amount_in: u128, amount_out:
           u128) {
77         if !is_promise_success() {
78             emit::ref_withdraw_failed(&dca_vault.user, amount_out, &dca_vault.token_out);
79             self.internal_unlock_dca_vault(dca_vault);
80             return;
81         }
82         emit::ref_withdraw_succeeded(&dca_vault.user, amount_out, &dca_vault.token_out);
83         // complete once dca
84         // calculate protocol fee
85         let (real_amount_out, protocol_fee) = self.internal_calculate_protocol_fee(amount_out);
86
87
88         self.internal_record_ref_normal(dca_vault, amount_in, real_amount_out);
89         // update asset
90         self.internal_increase_locked_assets(&dca_vault.user, &dca_vault.token_out, &U128::from(
                real_amount_out));
91         // update global asset
92         self.internal_increase_global_asset(&dca_vault.token_out, &U128::from(amount_out));
93         // add protocol asset
94         self.internal_increase_protocol_fee(&dca_vault.token_out, &U128::from(protocol_fee));
95     }

```

**Listing 2.29:** dca\_callback.rs

**Impact** Users have not deposited a storage fee for `token_out`.

**Suggestion** Add a check to ensure that users deposit a storage fee for `token_out` when creating a DCA.

## 2.1.14 Lack of lock during the withdrawals from Ref-Exchange

**Severity** High

**Status** Fixed in [Version 2](#)

**Introduced by** [Version 1](#)

**Description** During the execution of function `execute_dca()` process, users first deposit assets into `Ref-Exchange`, then perform a swap, and finally withdraw the `token_out` from `Ref-Exchange`. If the last step fails, the protocol sets the `DCA`'s status to `DCA_STATUS_SWAPPED`, indicating that the exchange has been completed, and in the next execution, only the asset withdrawal from `Ref-Exchange` is needed. However, when a user's `DCA` status is `DCA_STATUS_SWAPPED`, the user can invoke `execute_dca()` multiple times in a single block, and the function `internal_ref_withdraw()` will make a cross-contract invocation to `Ref-Exchange`'s function `withdraw()`. In this case, the protocol erroneously withdraws money from `Ref-Exchange` many times, which is incorrect.

```
64 pub fn execute_dca(&mut self, vault_id: String, swap_msg: String) {
65     require!(env::attached_deposit() == EXECUTE_DCA_FEE);
66     require!(self.status == DCAStatus::Running, PAUSE_OR_SHUTDOWN);
67     require!(self.market_user_map.get(&(env::predecessor_account_id())).unwrap(), INVALID_USER)
68     ;
69     require!(self.dca_vault_map.contains_key(&vault_id), INVALID_VAULT_ID);
70     let mut dca_vault = self.dca_vault_map.get(&vault_id).unwrap();
71     require!(!dca_vault.locked, LOCKED);
72     require!(!dca_vault.closed, DCA_CLOSED);
73     self.internal_check_dca_buy_available(&dca_vault);
74     if dca_vault.process == DCA_STATUS_SWAPPED {
75         self.internal_ref_withdraw(&mut dca_vault);
76         return;
77     }
78     let pair_path_key = self.internal_get_pair_key(&dca_vault.token_in, &dca_vault.token_out);
79     let path_op = self.recorded_pair_path.get(&pair_path_key);
80     // execute buy
81     if self.internal_check_need_oracle(&dca_vault) {
82         // require oracle
83         self.get_price_for_execute(&mut dca_vault, swap_msg);
84     } else if path_op.is_some() && path_op.clone().unwrap().len() > 0 {
85         // check mint amount out
86         let single_amount_out = dca_vault.single_amount_in.0;
87         self.internal_ref_estimate(&mut dca_vault, swap_msg, path_op.unwrap(), 0,
88             single_amount_out);
89     } else {
90         // direct buy
91         self.internal_execute_buy(&mut dca_vault, swap_msg, None, None);
92     }
93 }
```

Listing 2.30: dca.rs

```
71 pub fn internal_ref_withdraw(&mut self, dca_vault: &mut DCAVault) {
72     ext_ref::ext(self.ref_exchange_id.clone())
73         .with_attached_deposit(1)
74         .with_static_gas(GAS_FOR_WITHDRAW)
75         .withdraw(
76             dca_vault.token_out.clone(),
77             dca_vault.need_withdraw_amount.clone(),
78             None,
79             None
80         )
81 }
```

```

80         ).then(
81             Self::ext(env::current_account_id())
82                 .with_static_gas(GAS_FOR_WITHDRAW_CALL_BACK)
83                 .callback_do_withdraw(dca_vault, dca_vault.single_amount_in.0, dca_vault.
                        need_withdraw_amount.0)
84         );
85     }

```

**Listing 2.31:** dca\_private.rs

**Impact** The protocol can withdraw more assets than expected from Ref-Exchange.

**Suggestion** Revise the logic to ensure that the **DCA** is locked before the withdrawal process is completed.

## 2.2 Additional Recommendation

### 2.2.1 Redundant code

**Status** Confirmed

**Introduced by** Version 1

**Description** In the function `execute_dca()`, the check on line 65 is redundant; only the check on line 66 needs to be retained. The function `set_per_grid_storage_fee()` is not used anywhere in the protocol, and the `buy_amount_to_user` field in the `DCAVault` structure is also unused throughout the protocol.

```

9  pub fn create_dca(&mut self, name: String, token_in: AccountId, token_out: AccountId,
    single_amount_in: U128,
10      start_time: u64, interval_time: u64, count: u16, lowest_price: u64,
    highest_price: u64, slippage: u16) -> bool {
11      // record storage fee
12      let initial_storage_usage = env::storage_usage();
13      let user = env::predecessor_account_id();
14      require!(slippage >= MIN_SLIPPAGE, SLIPPAGE_TOO_SMALL);
15      require!(start_time > env::block_timestamp_ms(), INVALID_START_TIME);
16      require!(self.deposit_limit_map.contains_key(&token_in) && self.deposit_limit_map.
    contains_key(&token_out), INVALID_TOKEN);
17      if self.status != DCAStatus::Running {
18          self.internal_create_bot_refund_with_near(&user, &token_in, &token_out, env::
    attached_deposit(), PAUSE_OR_SHUTDOWN);
19          return false;
20      }
21      let total_amount_in = single_amount_in.0 * (count as u128);
22      if self.internal_get_user_balance(&user, &token_in).0 < total_amount_in {
23          self.internal_create_bot_refund_with_near(&user, &token_in, &token_out, env::
    attached_deposit(), LESS_TOKEN_IN);
24          return false;
25      }
26      // create id
27      let next_id = self.internal_get_and_use_next_id().to_string();
28      let next_dca_key = self.internal_get_dca_key(next_id);

```

```

29     let dca_vault = DCAVault {
30         name,
31         user: user.clone(),
32         id: next_dca_key.clone(),
33         closed: false,
34         token_in: token_in.clone(),
35         token_out,
36         start_time,
37         interval_time,
38         single_amount_in,
39         count,
40         execute_count: 0,
41         lowest_price,
42         highest_price,
43         left_amount_in: U128::from(total_amount_in),
44         buy_amount_record: U128::from(0),
45         slippage,
46         process: DCA_STATUS_NORMAL,
47         locked: false,
48         need_withdraw_amount: U128::from(0),
49         buy_amount_to_user: false,
50     };
51     self.dca_vault_map.insert(&next_dca_key, &dca_vault);
52     emit::create_dca(dca_vault);
53     // add locked asset
54     self.internal_transfer_assets_to_lock(&user, &token_in, U128::from(total_amount_in));
55
56
57     // refund storage fee
58     self.internal_refund_deposit(env::attached_deposit(), initial_storage_usage, &user);
59     return true;
60 }

```

**Listing 2.32:** dca.rs

```

11 pub fn internal_reduce_asset(&mut self, user: &AccountId, token: &AccountId, amount: &U256C) {
12     let mut user_balances = self.user_balances_map.get(user).unwrap_or_else(|| {
13         let mut map = LookupMap::new(StorageKey::UserBalanceSubKey(user.clone()));
14         map.insert(token, &U256C::from(0));
15         map
16     });
17
18
19     let balance = user_balances.get(token).unwrap_or(U256C::from(0));
20     user_balances.insert(token, &(balance - amount));
21
22
23     self.user_balances_map.insert(user, &user_balances);
24 }

```

**Listing 2.33:** grid\_bot\_asset.rs

```

164 pub fn set_per_grid_storage_fee(&mut self, new_per_grid_storage_fee: U128) {

```



```

165     self.assert_owner();
166     self.per_grid_storage_fee = new_per_grid_storage_fee.0;
167 }

```

**Listing 2.34:** dca\_owner.rs

```

62 pub fn execute_dca(&mut self, vault_id: String, swap_msg: String) {
63     require!(env::attached_deposit() == EXECUTE_DCA_FEE);
64     require!(self.status == DCASatus::Running, PAUSE_OR_SHUTDOWN);
65     require!(self.market_user_map.contains_key(&(env::predecessor_account_id())), INVALID_USER)
66     ;
67     require!(self.market_user_map.get(&(env::predecessor_account_id())).unwrap(), INVALID_USER)
68     ;
69     require!(self.dca_vault_map.contains_key(&vault_id), INVALID_VAULT_ID);
70     let mut dca_vault = self.dca_vault_map.get(&vault_id).unwrap();
71     require!(!dca_vault.locked, LOCKED);
72     require!(!dca_vault.closed, DCA_CLOSED);
73     self.internal_check_dca_buy_available(&dca_vault);
74     if dca_vault.process == DCA_STATUS_SWAPPED {
75         self.internal_ref_withdraw(&mut dca_vault);
76         return;
77     }
78     let pair_key = self.internal_get_pair_key(&dca_vault.token_in, &dca_vault.token_out);
79     let path_op = self.recorded_pair_path.get(&pair_key);
80     // execute buy
81     if self.internal_check_need_oracle(&dca_vault) {
82         // require oracle
83         self.get_price_for_execute(&mut dca_vault, swap_msg);
84     } else if path_op.is_some() && path_op.clone().unwrap().len() > 0 {
85         // check mint amount out
86         let single_amount_out = dca_vault.single_amount_in.0;
87         self.internal_ref_estimate(&mut dca_vault, swap_msg, path_op.unwrap(), 0,
88             single_amount_out);
89     } else {
90         // direct buy
91         self.internal_execute_buy(&mut dca_vault, swap_msg, None, None);
92     }
93 }

```

**Listing 2.35:** dca.rs

**Suggestion** Remove this redundant code.

**Feedback from the project** The parameter `buy_amount_to_user` is considered in anticipation that if the `Ref-Exchange` contract is modified to support swapping to a specified address, it would facilitate an easy upgrade to this mode.

## 2.2.2 Standardize owner checks with `assert_owner_without_yocto()`

**Status** Fixed in [Version 2](#)

**Introduced by** [Version 1](#)

**Description** In the functions `register_pair()`, `set_min_deposit()`, `storage_deposit()`, `enable_oracle_config()`, and `set_market_user()`, the current implementation uses `require!(env::predecessor_account_id() == self.owner_id, ERR_NOT_ALLOWED)` to verify whether the invoker is the `owner`. However, since the contract already implements the function `assert_owner()_without_yocto()`, it is recommended to replace these checks with the function `assert_owner()_without_yocto()` to streamline and unify the ownership verification process across the contract.

```
145 pub fn set_market_user(&mut self, market_user: AccountId, enable: bool) {
146     require!(env::predecessor_account_id() == self.owner_id, ERR_NOT_ALLOWED);
147     require!(env::attached_deposit() == DEFAULT_CONFIG_SET_STORAGE_FEE, LESS_STORAGE_FEE);
148     self.market_user_map.insert(&market_user, &enable);
149 }
```

**Listing 2.36:** dca.rs

```
81 pub fn register_pair(&mut self, token_a: AccountId, token_b: AccountId, token_a_min_deposit:
    U128, token_b_min_deposit: U128, token_a_oracle_id_op: Option<String>,
    token_b_oracle_id_op: Option<String>, path: Vec<Pool>) {
82     require!(env::predecessor_account_id() == self.owner_id, ERR_NOT_ALLOWED);
83     require!(env::attached_deposit() == REGISTER_PAIR_STORAGE_FEE * 2, LESS_STORAGE_FEE);
84     require!(token_a == path.get(0).unwrap().token_in && token_b == path.get(path.len() - 1).
        unwrap().token_out, INVALID_TOKEN);
85
86
87     let pair_key = self.internal_get_pair_key(&token_a, &token_b);
88     // record Pair pool id
89     self.recorded_pair_path.insert(&pair_key, &path);
90
91
92     self.deposit_limit_map.insert(&token_a, &token_a_min_deposit);
93     self.deposit_limit_map.insert(&token_b, &token_b_min_deposit);
94
95
96     self.internal_increase_global_asset(&token_a, &U128::from(0));
97     self.internal_increase_global_asset(&token_b, &U128::from(0));
98
99
100    self.internal_increase_protocol_fee(&token_a, &U128::from(0));
101    self.internal_increase_protocol_fee(&token_b, &U128::from(0));
102
103
104    self.internal_increase_locked_in_ref_asset(&token_a, &U128::from(0));
105    self.internal_increase_locked_in_ref_asset(&token_b, &U128::from(0));
106
107
108    self.internal_set_oracle(&token_a, token_a_oracle_id_op);
109    self.internal_set_oracle(&token_b, token_b_oracle_id_op);
110
111
112    self.internal_storage_deposit(&env::current_account_id(), &token_a,
        REGISTER_TOKEN_STORAGE_FEE);
```

```

113     self.internal_storage_deposit(&env::current_account_id(), &token_b,
114                                   REGISTER_TOKEN_STORAGE_FEE);
115
116     self.internal_ref_storage_deposit(&env::current_account_id(), REGISTER_TOKEN_STORAGE_FEE);
117 }
118
119
120 #[payable]
121 pub fn set_min_deposit(&mut self, token: AccountId, min_deposit: U128) {
122     require!(env::predecessor_account_id() == self.owner_id, ERR_NOT_ALLOWED);
123     require!(env::attached_deposit() == DEFAULT_CONFIG_SET_STORAGE_FEE, LESS_STORAGE_FEE);
124     self.deposit_limit_map.insert(&token, &min_deposit);
125 }
126
127
128 #[payable]
129 pub fn storage_deposit(&mut self, token: AccountId, storage_fee: U128) {
130     require!(env::predecessor_account_id() == self.owner_id, ERR_NOT_ALLOWED);
131     require!(env::attached_deposit() == storage_fee.0, LESS_TOKEN_STORAGE_FEE);
132     self.internal_storage_deposit(&env::current_account_id(), &token, storage_fee.0);
133 }
134
135
136 #[payable]
137 pub fn enable_oracle_config(&mut self, token: AccountId, oracle_id: String) {
138     require!(env::predecessor_account_id() == self.owner_id, ERR_NOT_ALLOWED);
139     require!(env::attached_deposit() == DEFAULT_CONFIG_SET_STORAGE_FEE, LESS_STORAGE_FEE);
140     self.internal_set_oracle(&token, Some(oracle_id));
141 }

```

**Listing 2.37:** dca.rs

**Suggestion** Replace `owner` checks with function `assert_owner()_without_yocto`.

### 2.2.3 Lack of setting static gas

**Status** Fixed in [Version 2](#)

**Introduced by** [Version 1](#)

**Description** The function `withdraw_near()` makes a cross-contract invocation to the `WNEAR`'s function `near_withdraw()`, but does not specify a static gas for the cross-contract invocation. This also occurs in the function `private_create_pair_price_request()` at line 103.

```

15 pub fn withdraw_near(&mut self, user: &AccountId, amount: u128) {
16     ext_wnear::ext(self.wnear.clone())
17         .with_attached_deposit(ONE_YOCTO)
18         .near_withdraw(U128::from(amount))
19         .then(
20             Self::ext(env::current_account_id())
21                 .after_withdraw_near(
22                     user,
23                     amount,

```

```
24         )
25     );
26 }
```

**Listing 2.38:** `wnear.rs`

```
98 fn private_create_pair_price_request(&self, token_in: &AccountId, token_out: &AccountId) -> (
    Promise, Vec<AccountId>) {
99     let token_in_id = self.oracle_map.get(token_in).unwrap();
100    let token_out_id = self.oracle_map.get(token_out).unwrap();
101    let identifiers = vec![self.internal_format_price_identifier(token_in_id), self.
        internal_format_price_identifier(token_out_id)];
102    let tokens = vec![token_in.clone(), token_out.clone()];
103    let mut promise = ext_pyth::ext(self.oracle.clone()).get_price(identifiers[0].clone());
104    for index in 1..identifiers.len() {
105        promise = promise.and(ext_pyth::ext(self.oracle.clone()).with_static_gas(
            GAS_FOR_GET_ORACLE_PRICE).get_price(identifiers[index].clone()));
106    }
107    return (promise, tokens);
108 }
```

**Listing 2.39:** `oracle.rs`

**Suggestion** Set sufficient gas for cross-contract invokes to ensure they do not fail due to insufficient gas.

#### 2.2.4 Lack of check in function `token_storage_deposit()`

**Status** Confirmed

**Introduced by** [Version 1](#)

**Description** Users can deposit a storage fee for a token using the function `token_storage_deposit()`. However, the function does not check whether the storage fee has been deposited for the token. This could lead to unnecessary loss.

```
121 pub fn token_storage_deposit(&mut self, user: AccountId, token: AccountId) {
122     require!(env::attached_deposit() == BASE_CREATE_STORAGE_FEE);
123     require!(self.deposit_limit_map.contains_key(&token), INVALID_TOKEN);
124     let initial_storage_usage = env::storage_usage();
125     self.internal_register_token_for_user(&user, &token);
126     self.internal_refund_deposit(BASE_CREATE_STORAGE_FEE, initial_storage_usage, &env::
        predecessor_account_id());
127 }
```

**Listing 2.40:** `dca.rs`

**Suggestion** Add a check to ensure that the user has not previously deposited a storage fee for the specified token.

**Feedback from the project** There is storage fee refund logic.

## 2.2.5 Incorrect gas calculation

**Status** Confirmed

**Introduced by** Version 1

**Description** In the function `internal_ref_estimate()`, the final amount of `token_out` obtained from the exchange is calculated by traversing the swap path. During the first iteration, the gas for function `get_return()` is allocated as `GAS_FOR_REF_ESTIMATE_ONCE`. However, since the `current_path` index is 0 at this time, the gas used during the first iteration is not subtracted when calculating the gas to be set for the callback function in subsequent iterations.

```

47 pub fn internal_ref_estimate(&mut self, dca_vault: &mut DCAVault, swap_msg: String, path: Vec<
    Pool>, current_path: u8, amount_in: u128) -> Promise {
48     let pool = path.get(current_path as usize).unwrap();
49     ext_ref::ext(self.ref_exchange_id.clone())
50         .with_static_gas(GAS_FOR_REF_ESTIMATE_ONCE)
51         .get_return(pool.pool_id.clone(), pool.token_in.clone(), U128::from(amount_in), pool.
            token_out.clone())
52         .then(
53             Self::ext(env::current_account_id())
54                 .with_static_gas(Gas(GAS_FOR_REF_ESTIMATE.0 - GAS_FOR_REF_ESTIMATE_ONCE.0 * (
                    current_path as u64)))
55                 .callback_ref_estimate(
56                     dca_vault,
57                     swap_msg,
58                     path,
59                     current_path,
60                     amount_in,
61                 )
62         )
63     }

```

**Listing 2.41:** refexchange.rs

**Suggestion** Revise the logic accordingly.

**Feedback from the project** The 280 gas already takes into account the first 5 gas consumption.

## 2.3 Note

### 2.3.1 Decision of swap path in function `execute_dca()`

**Introduced by** Version 1

**Description** Any user can create a `DCA`, and then a `maker` can utilize the `DCA` created by the user to assist in executing periodic investments. The execution sequence begins by depositing the user's assets into the `Ref-Exchange`, followed by invoking the function `execute_actions()` of the `Ref-Exchange` to complete the exchange. If the token being exchanged does not rely on an oracle, and a swap path from `token_in` to `token_out` is recorded in the protocol, it enters the function `internal_ref_estimate()`. This function traverses the pools along the path to calculate the final amount of `token_out` exchanged. However, in the function `execute_actions()`,

the path used is derived from the `swap_info` even if the swap path from `token_in` to `token_out` is listed in the protocol's whitelist.

**Feedback from the project** The design is intentional. Initially, there was no `Ref-Exchange` estimation, which required users to fully trust our Operator role, posing a significant risk. To mitigate this risk, we introduced `Ref-Exchange` estimation, essentially using `Ref-Exchange` as an oracle. The configured pool paths will only involve one or two pools, not the optimal router, and if the project team withdraws liquidity from the pools, it would lead to losses for users executing DCA, hence the decision not to use this path.

### 2.3.2 Potential centralization risk

**Introduced by** `Version 1`

**Description** The protocol includes several privileged functions, such as `register_pair()`, which can arbitrarily register tokens, and `enable_oracle_config()`, which can arbitrarily set oracles. If the `owner`'s private key is lost or maliciously exploited, it could potentially cause losses to users.

