Security Audit Report for DeltaTrade

Date: September 10, 2024 **Version:** 1.0
Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	DeltaTrade
Target	DeltaTrade

Version History

Version	Date	Description
1.0	September 10, 2024	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-09-10 07:57:28+0000

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[1] 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	`Version 1`	`d3411c3e01d1d96b5fdeff7b4e82e58dac1c433b`
	`Version 2`	`6195858093c49d8d651e9c79e51b427119301dce`

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.

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

17	-	Lack of setting static gas	Recommendation	Fixed
18	-	Lack of check in function `token_storage_deposit()`	Recommendation	Confirmed
19	-	Incorrect gas calculation	Recommendation	Confirmed
20	-	Decision of swap path in function `execute_dca()`	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.

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

Listing 2.1: dca_owner.rs

```rust
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>) {
    require!(env::predecessor_account_id() == self.owner_id, ERR_NOT_ALLOWED);
    require!(env::attached_deposit() == REGISTER_PAIR_STORAGE_FEE * 2, LESS_STORAGE_FEE);
    require!(token_a == path.get(0).unwrap().token_in && token_b == path.get(path.len() - 1).
        unwrap().token_out, INVALID_TOKEN);


    let pair_key = self.internal_get_pair_key(&token_a, &token_b);
    // record Pair pool id
    self.recorded_pair_path.insert(&pair_key, &path);


    self.deposit_limit_map.insert(&token_a, &token_a_min_deposit);
    self.deposit_limit_map.insert(&token_b, &token_b_min_deposit);


    self.internal_increase_global_asset(&token_a, &U128::from(0));
    self.internal_increase_global_asset(&token_b, &U128::from(0));


    self.internal_increase_protocol_fee(&token_a, &U128::from(0));
    self.internal_increase_protocol_fee(&token_b, &U128::from(0));


    self.internal_increase_locked_in_ref_asset(&token_a, &U128::from(0));
    self.internal_increase_locked_in_ref_asset(&token_b, &U128::from(0));


    self.internal_set_oracle(&token_a, token_a_oracle_id_op);
    self.internal_set_oracle(&token_b, token_b_oracle_id_op);


    self.internal_storage_deposit(&env::current_account_id(), &token_a,
        REGISTER_TOKEN_STORAGE_FEE);
    self.internal_storage_deposit(&env::current_account_id(), &token_b,
        REGISTER_TOKEN_STORAGE_FEE);


    self.internal_ref_storage_deposit(&env::current_account_id(), REGISTER_TOKEN_STORAGE_FEE);
}
```

Listing 2.2: dca_owner.rs

```
pub fn create_dca(&mut self, name: String, token_in: AccountId, token_out: AccountId,
    single_amount_in: U128,
                start_time: u64, interval_time: u64, count: u16, lowest_price: u64,
                    highest_price: u64, slippage: u16) -> bool {
    // record storage fee
    let initial_storage_usage = env::storage_usage();
    let user = env::predecessor_account_id();
    require!(slippage >= MIN_SLIPPAGE, SLIPPAGE_TOO_SMALL);
    require!(start_time > env::block_timestamp_ms(), INVALID_START_TIME);
    require!(self.deposit_limit_map.contains_key(&token_in) && self.deposit_limit_map.
        contains_key(&token_out), INVALID_TOKEN);
    if self.status != DCAStatus::Running {
        self.internal_create_bot_refund_with_near(&user, &token_in, &token_out, env::
            attached_deposit(), PAUSE_OR_SHUTDOWN);
        return false;
    }
    let total_amount_in = single_amount_in.0 * (count as u128);
    if self.internal_get_user_balance(&user, &token_in).0 < total_amount_in {
        self.internal_create_bot_refund_with_near(&user, &token_in, &token_out, env::
            attached_deposit(), LESS_TOKEN_IN);
        return false;
    }
    // create id
    let next_id = self.internal_get_and_use_next_id().to_string();
    let next_dca_key = self.internal_get_dca_key(next_id);
    let dca_vault = DCAVault {
        name,
        user: user.clone(),
        id: next_dca_key.clone(),
        closed: false,
        token_in: token_in.clone(),
        token_out,
        start_time,
        interval_time,
        single_amount_in,
        count,
        execute_count: 0,
        lowest_price,
        highest_price,
        left_amount_in: U128::from(total_amount_in),
        buy_amount_record: U128::from(0),
        slippage,
        process: DCA_STATUS_NORMAL,
        locked: false,
        need_withdraw_amount: U128::from(0),
        buy_amount_to_user: false,
    };
    self.dca_vault_map.insert(&next_dca_key, &dca_vault);
    emit::create_dca(dca_vault);
    // add locked asset
    self.internal_transfer_assets_to_lock(&user, &token_in, U128::from(total_amount_in));


    // refund storage fee
    self.internal_refund_deposit(env::attached_deposit(), initial_storage_usage, &user);
    return true;
}
```

Listing 2.3: dca.rs

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

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.

```
#[payable]
pub fn execute_dca(&mut self, vault_id: String, swap_msg: String) {
    require!(env::attached_deposit() == EXECUTE_DCA_FEE);
    require!(self.status == DCAStatus::Running, PAUSE_OR_SHUTDOWN);
    require!(self.market_user_map.contains_key(&(env::predecessor_account_id())), INVALID_USER)
        ;
    require!(self.market_user_map.get(&(env::predecessor_account_id())).unwrap(), INVALID_USER)
        ;
    require!(self.dca_vault_map.contains_key(&vault_id), INVALID_VAULT_ID);
    let mut dca_vault = self.dca_vault_map.get(&vault_id).unwrap();
    require!(!dca_vault.locked, LOCKED);
    require!(!dca_vault.closed, DCA_CLOSED);
    self.internal_check_dca_buy_available(&dca_vault);
    if dca_vault.process == DCA_STATUS_SWAPPED {
        self.internal_ref_withdraw(&mut dca_vault);
        return;
    }
    let pair_key = self.internal_get_pair_key(&dca_vault.token_in, &dca_vault.token_out);
    let path_op = self.recorded_pair_path.get(&pair_key);
    // execute buy
    if self.internal_check_need_oracle(&dca_vault) {
        // require oracle
        self.get_price_for_execute(&mut dca_vault, swap_msg);
    } else if path_op.is_some() && path_op.clone().unwrap().len() > 0 {
        // check mint amount out
        let single_amount_out = dca_vault.single_amount_in.0;
        self.internal_ref_estimate(&mut dca_vault, swap_msg, path_op.unwrap(), 0,
            single_amount_out);
    } else {
        // direct buy
        self.internal_execute_buy(&mut dca_vault, swap_msg, None, None);
    }
}
```

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.

```rust
#[payable]
pub fn close_dca(&mut self, vault_id: String) {
    assert_one_yocto();
    require!(self.dca_vault_map.contains_key(&vault_id), VAULT_NOT_EXIST);
    let mut dca_vault = self.dca_vault_map.get(&vault_id).unwrap();
    require!(!dca_vault.closed, INVALID_BOT_STATUS);
    // check permission, user self close
    require!(env::predecessor_account_id() == dca_vault.user, INVALID_USER);


    dca_vault.closed = true;
    self.internal_transfer_assets_to_unlock(&(dca_vault.user), &(dca_vault.token_in), dca_vault
        .left_amount_in.clone());
    self.internal_transfer_assets_to_unlock(&(dca_vault.user), &(dca_vault.token_out),
        dca_vault.buy_amount_record.clone());
    // update dca_vault info
    self.dca_vault_map.insert(&vault_id, &dca_vault);
    // withdraw
    self.internal_withdraw_all(&(dca_vault.user), &(dca_vault.token_in));
    self.internal_withdraw_all(&(dca_vault.user), &(dca_vault.token_out));


    emit::close_dca(&env::predecessor_account_id(), dca_vault.id.clone(), dca_vault.
        left_amount_in.0, dca_vault.buy_amount_record.0);
}
```

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.

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

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.

```rust
#[private]
pub fn callback_do_withdraw(&mut self, dca_vault: &mut DCAVault, amount_in: u128, amount_out:
    u128) {
    if !is_promise_success() {
        emit::ref_withdraw_failed(&dca_vault.user, amount_out, &dca_vault.token_out);
        self.internal_unlock_dca_vault(dca_vault);
        return;
    }
    emit::ref_withdraw_succeeded(&dca_vault.user, amount_out, &dca_vault.token_out);
    // complete once dca
    // calculate protocol fee
    let (real_amount_out, protocol_fee) = self.internal_calculate_protocol_fee(amount_out);


    self.internal_record_ref_normal(dca_vault, amount_in, real_amount_out);
    // update asset
    self.internal_increase_locked_assets(&dca_vault.user, &dca_vault.token_out, &U128::from(
        real_amount_out));
    // update global asset
    self.internal_increase_global_asset(&dca_vault.token_out, &U128::from(amount_out));
    // add protocol asset
    self.internal_increase_protocol_fee(&dca_vault.token_out, &U128::from(protocol_fee));
}
```

Listing 2.8: dca_callback.rs

```rust
#[private]
pub fn exchange_callback_post_withdraw(
    &mut self,
    token_id: AccountId,
    sender_id: AccountId,
    amount: U128,
) -> U128 {
        assert_eq!(
            env::promise_results_count(),
            1,
            "{}",
            ERR25_CALLBACK_POST_WITHDRAW_INVALID
        );
        match env::promise_result(0) {
            PromiseResult::NotReady => unreachable!(),
            PromiseResult::Successful(_) => amount,
            PromiseResult::Failed => {
                // This reverts the changes from withdraw function.
                // If account doesn't exit, deposits to the owner's account as lostfound.
                let mut failed = false;
                if let Some(mut account) = self.internal_get_account(&sender_id) {
                    if account.deposit_with_storage_check(&token_id, amount.0) {
                        // cause storage already checked, here can directly save
                        self.accounts.insert(&sender_id, &account.into());
                    } else {
                        // we can ensure that internal_get_account here would NOT cause a version
                            upgrade,
                        // cause it is callback, the account must be the current version or non-
                            exist,
                        // so, here we can just leave it without insert, won't cause storage
                            collection inconsistency.
                        env::log(
                            format!(
                                "Account {} has not enough storage. Depositing to owner.",
                                sender_id
                            )
                            .as_bytes(),
                        );
                        failed = true;
                    }
                } else {
                    env::log(
                        format!(
                            "Account {} is not registered. Depositing to owner.",
                            sender_id
                        )
                        .as_bytes(),
                    );
                    failed = true;
                }
                if failed {
                    self.internal_lostfound(&token_id, amount.0);
                }
                0.into()
            }
        }
    }
```

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.

```rust
#[private]
pub fn callback_do_withdraw(&mut self, dca_vault: &mut DCAVault, amount_in: u128, amount_out:
    u128) {
    if !is_promise_success() {
        emit::ref_withdraw_failed(&dca_vault.user, amount_out, &dca_vault.token_out);
        self.internal_unlock_dca_vault(dca_vault);
        return;
    }
    emit::ref_withdraw_succeeded(&dca_vault.user, amount_out, &dca_vault.token_out);
    // complete once dca
    // calculate protocol fee
    let (real_amount_out, protocol_fee) = self.internal_calculate_protocol_fee(amount_out);


    self.internal_record_ref_normal(dca_vault, amount_in, real_amount_out);
    // update asset
    self.internal_increase_locked_assets(&dca_vault.user, &dca_vault.token_out, &U128::from(
        real_amount_out));
    // update global asset
    self.internal_increase_global_asset(&dca_vault.token_out, &U128::from(amount_out));
    // add protocol asset
    self.internal_increase_protocol_fee(&dca_vault.token_out, &U128::from(protocol_fee));
}
```

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.

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

Listing 2.11: dca_owner.rs

```rust
pub fn internal_ref_withdraw_directly(&mut self, user: &AccountId, token: &AccountId, amount:
    &U128) {
    ext_ref::ext(self.ref_exchange_id.clone())
        .with_attached_deposit(1)
        .with_static_gas(GAS_FOR_WITHDRAW)
        .withdraw(
            token.clone(),
            amount.clone(),
            None,
            None
        ).then(
        Self::ext(env::current_account_id())
            .with_static_gas(GAS_FOR_WITHDRAW_CALL_BACK)
            .callback_do_withdraw_directly(user, token, amount)
    );
}
```

Listing 2.12: dca_private.rs

```rust
pub fn withdraw(
    &mut self,
    token_id: ValidAccountId,
    amount: U128,
    unregister: Option<bool>,
    skip_unwrap_near: Option<bool>
) -> Promise {
    assert_one_yocto();
    self.assert_contract_running();
    let token_id: AccountId = token_id.into();
    // feature frozenlist
    self.assert_no_frozen_tokens(&[token_id.clone()]);
    let sender_id = env::predecessor_account_id();
    let mut account = self.internal_unwrap_account(&sender_id);

    // get full amount if amount param is 0
    let mut amount: u128 = amount.into();
    if amount == 0 {
        amount = account.get_balance(&token_id).expect(ERR21_TOKEN_NOT_REG);
    }
    assert!(amount > 0, "{}", ERR29_ILLEGAL_WITHDRAW_AMOUNT);

    // Note: subtraction and deregistration will be reverted if the promise fails.
    account.withdraw(&token_id, amount);
    if unregister == Some(true) {
        account.unregister(&token_id);
    }
    self.internal_save_account(&sender_id, account);
    self.internal_send_tokens(&sender_id, &token_id, amount, skip_unwrap_near)
}
```

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.

```
pub fn internal_execute_buy(&mut self, dca_vault: &mut DCAVault, swap_msg: String,
     price_list_op: Option<Vec<Price>>, estimate_amount_out_op: Option<u128>) -> bool {
    require!(self.status == DCAStatus::Running, PAUSE_OR_SHUTDOWN);
    require!(!dca_vault.locked, LOCKED);
    require!(!dca_vault.closed, DCA_CLOSED);
    require!(self.internal_get_user_locked_balance(&dca_vault.user, &dca_vault.token_in).0 >=
        dca_vault.single_amount_in.0, LESS_TOKEN_IN);
    let swap_info = serde_json::from_str::<SwapMessage>(&swap_msg).expect(INVALID_EXECUTE_MSG);
    // check swap param
    let (amount_in, amount_out) = self.internal_check_swap_info(&swap_info, &dca_vault);
    // check oracle
    self.internal_check_oracle_price(&dca_vault, price_list_op, amount_in, amount_out);
    // check estimate amount out
    self.internal_check_estimate_amount_out(amount_out, estimate_amount_out_op, dca_vault.
        slippage);


    self.internal_lock_dca_vault(dca_vault);
    if dca_vault.process == DCA_STATUS_NORMAL {
        // 1.deposit,2.swap,3.withdraw
        self.internal_ref_deposit(dca_vault, amount_in, &swap_info);
    } else if dca_vault.process == DCA_STATUS_DEPOSITED {
        // 1.swap, 2.withdraw
        self.internal_ref_swap(dca_vault, amount_in, &swap_info);
    }
    // else {
    //     // 1.withdraw
    //     self.internal_ref_withdraw(dca_vault);
    // }
    return true;
}
```

Listing 2.14: dca_private.rs

```
#[private]
pub fn callback_do_deposit(&mut self, dca_vault: &mut DCAVault, amount_in: u128, swap_info: &SwapMessage) {
    if !is_promise_success() {
        // deposit error, assets still on DCA contract, so don't need to do anything
        emit::ref_deposit_failed(&dca_vault.user, amount_in, &dca_vault.token_in);
        self.internal_unlock_dca_vault(dca_vault);
        return;
    }
    let cross_call_result = promise_result_as_success().expect(ERR102_CROSS_CONTRACT_FAILED);
    let amount_in_real = serde_json::from_slice::<U128>(&cross_call_result).unwrap().0;
    if amount_in != amount_in_real {
        // deposit must same as the total
        emit::ref_deposit_failed(&dca_vault.user, amount_in, &dca_vault.token_in);
        self.internal_unlock_dca_vault(dca_vault);
        if amount_in_real == 0 {
            // nothing to do, deposit 0, so don't need to do anything
            return;
        }
        // need owner to withdraw from ref
        emit::need_owner_withdraw_from_ref(&dca_vault.user, amount_in_real, &dca_vault.token_in);
        // add ref locked asset
        self.internal_increase_locked_in_ref_asset(&dca_vault.token_in, &U128::from(amount_in_real));
        // reduce asset from global
        // self.internal_reduce_locked_assets(&dca_vault.user, &dca_vault.token_in, &U128::from(amount_in));
        self.internal_reduce_global_asset(&dca_vault.token_in, &U128::from(amount_in_real));
        // withdraw
        self.internal_ref_withdraw_directly(&dca_vault.user, &dca_vault.token_in, &U128::from(amount_in_real));
        return;
    }
```

```
        emit::ref_deposit_success(&dca_vault.user, amount_in, &dca_vault.token_in);
        // reduce asset from user locked asset and global
        self.internal_reduce_locked_assets(&dca_vault.user, &dca_vault.token_in, &U128::from(
            amount_in));
        self.internal_reduce_global_asset(&dca_vault.token_in, &U128::from(amount_in_real));
        // execute deposit record
        self.internal_record_ref_deposited(dca_vault);
        // execute swap
        self.internal_ref_swap(dca_vault, amount_in, swap_info);
    }
```

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`.

```
    pub fn internal_check_swap_info(&self, swap_info: &SwapMessage, dca_vault: &DCAVault) -> (u128
        , u128) {
        let mut amount_in = 0;
        let mut amount_out = 0;
        require!(!swap_info.actions.is_empty(), INVALID_SWAP_ACTIONS);


        let mut pre_action: Option<&Action> = None;
        for (index, action) in swap_info.actions.iter().enumerate() {
            if index == 0 {
                require!(action.token_in == dca_vault.token_in, INVALID_TOKEN_IN);
            }
            if index == swap_info.actions.len() - 1 {
                require!(action.token_out == dca_vault.token_out, INVALID_TOKEN_OUT);
            }
            if action.token_in == dca_vault.token_in {
                amount_in += action.amount_in.expect(ERR100_WRONG_MSG_FORMAT).0
            }
            if action.token_out == dca_vault.token_out {
                amount_out += action.min_amount_out.0;
            }
            if pre_action.clone().is_some() && action.token_in != dca_vault.token_in {
                // must be a chain
```

```rust
            require!(pre_action.clone().unwrap().token_out == action.token_in,
                INVALID_SWAP_CHAIN_PRE_ACTION_OUT_MUST_ACTION_IN);
            require!(action.amount_in.is_none(), INVALID_AMOUNT_IN);
        }
        if pre_action.clone().is_some() && action.token_in == dca_vault.token_in {
            require!(pre_action.clone().unwrap().token_out == dca_vault.token_out,
                INVALID_SWAP_CHAIN_PRE_ACTION_OUT_MUST_ACTION_OUT);
            require!(action.amount_in.is_some(), INVALID_AMOUNT_IN);
        }
        pre_action = Some(action);
    }
    require!(amount_in == dca_vault.single_amount_in.0, INVALID_AMOUNT_IN);
    return (amount_in, amount_out);
}
```

Listing 2.16: dca_check.rs

```rust
#[private]
pub fn callback_do_swap(&mut self, dca_vault: &mut DCAVault, amount_in: u128) {
    if !is_promise_success() {
        emit::ref_swap_failed(&dca_vault.user, amount_in, &dca_vault.token_in, &dca_vault.
            token_out);
        self.internal_unlock_dca_vault(dca_vault);
        return;
    }


    let cross_call_result = promise_result_as_success().expect(ERR102_CROSS_CONTRACT_FAILED);
    let action_result = serde_json::from_slice::<ActionResult>(&cross_call_result).unwrap();
    match action_result {
        ActionResult::None => {
            emit::ref_swap_failed(&dca_vault.user, amount_in, &dca_vault.token_in, &dca_vault.
                token_out);
            self.internal_unlock_dca_vault(dca_vault);
        }
        ActionResult::Amount(amount) => {
            emit::ref_swap_success(&dca_vault.user, amount_in, amount.0, &dca_vault.token_in, &
                dca_vault.token_out);
            // record to swapped
            self.internal_record_ref_swapped(dca_vault, amount.clone());
            // withdraw
            self.internal_ref_withdraw(dca_vault);
        }
    }
}
```

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.

```rust
pub fn internal_check_oracle_price(&self, dca_vault: &DCAVault, price_list_op: Option<Vec<
    Price>>, amount_in: u128, amount_out: u128) {
    let in_meta_decimal = self.token_decimal_map.get(&dca_vault.token_in).unwrap();
    let out_meta_decimal = self.token_decimal_map.get(&dca_vault.token_out).unwrap();
    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();
    self.internal_check_price_limit(swap_price, &dca_vault);
    if !self.internal_check_need_oracle(&dca_vault) {
        return;
    }
    let price_list = price_list_op.unwrap();
    let in_price = &price_list[0];
    let out_price = &price_list[1];
    require!(in_price.publish_time as u64 * 1000 + self.oracle_valid_time.clone() >= env::
        block_timestamp_ms(), PRICE_EXPIRED);
    require!(out_price.publish_time as u64 * 1000 + self.oracle_valid_time.clone() >= env::
        block_timestamp_ms(), PRICE_EXPIRED);
    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);
    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);

    let oracle_price = (in_scaled_price * BigDecimal::from(PRICE_DENOMINATOR) /
        out_scaled_price).round_down_u128();
    self.internal_check_price_limit(oracle_price, &dca_vault);

    if swap_price >= oracle_price {
        require!((swap_price - oracle_price) * SLIPPAGE_DENOMINATOR as u128 / swap_price <=
            dca_vault.slippage as u128, OUT_OF_SLIPPAGE);
    } else {
        require!((oracle_price - swap_price) * SLIPPAGE_DENOMINATOR as u128 / swap_price <=
            dca_vault.slippage as u128, OUT_OF_SLIPPAGE);
    }
}
```

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.

```rust
pub fn internal_check_swap_info(&self, swap_info: &SwapMessage, dca_vault: &DCAVault) -> (u128
    , u128) {
    let mut amount_in = 0;
    let mut amount_out = 0;
    require!(!swap_info.actions.is_empty(), INVALID_SWAP_ACTIONS);


    let mut pre_action: Option<&Action> = None;
    for (index, action) in swap_info.actions.iter().enumerate() {
        if index == 0 {
            require!(action.token_in == dca_vault.token_in, INVALID_TOKEN_IN);
        }
        if index == swap_info.actions.len() - 1 {
            require!(action.token_out == dca_vault.token_out, INVALID_TOKEN_OUT);
        }
        if action.token_in == dca_vault.token_in {
            amount_in += action.amount_in.expect(ERR100_WRONG_MSG_FORMAT).0
        }
        if action.token_out == dca_vault.token_out {
            amount_out += action.min_amount_out.0;
        }
        if pre_action.clone().is_some() && action.token_in != dca_vault.token_in {
            // must be a chain
            require!(pre_action.clone().unwrap().token_out == action.token_in,
                INVALID_SWAP_CHAIN_PRE_ACTION_OUT_MUST_ACTION_IN);
            require!(action.amount_in.is_none(), INVALID_AMOUNT_IN);
        }
        if pre_action.clone().is_some() && action.token_in == dca_vault.token_in {
            require!(pre_action.clone().unwrap().token_out == dca_vault.token_out,
                INVALID_SWAP_CHAIN_PRE_ACTION_OUT_MUST_ACTION_OUT);
            require!(action.amount_in.is_some(), INVALID_AMOUNT_IN);
        }
        pre_action = Some(action);
    }
    require!(amount_in == dca_vault.single_amount_in.0, INVALID_AMOUNT_IN);
    return (amount_in, amount_out);
}
```

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.

```
pub fn withdraw_near(&mut self, user: &AccountId, amount: u128) {
    ext_wnear::ext(self.wnear.clone())
        .with_attached_deposit(ONE_YOCTO)
        .near_withdraw(U128::from(amount))
        .then(
            Self::ext(env::current_account_id())
                .after_withdraw_near(
                    user,
                    amount,
                )
        );
}
```

Listing 2.20: wnear.rs

```
#[private]
fn after_withdraw_near(&mut self, user: &AccountId, amount: u128) -> bool {
    let promise_success = is_promise_success();
    if !promise_success.clone() {
        emit::wrap_near_error(user, 0, amount, false);
        self.internal_increase_asset(user, &self.wnear.clone(), &(U128::from(amount)));
    } else {
        self.internal_ft_transfer_near(user, amount, true);
    }
    promise_success
}
```

Listing 2.21: wnear.rs

```
pub fn internal_ft_transfer_near(&mut self, receiver_id: &AccountId, amount: Balance,
     effect_global_balance: bool) -> Promise {
        require!(self.internal_get_remaining_gas() >= GAS_FOR_FT_TRANSFER, LESS_GAS);
        if effect_global_balance {
            // reduce from global asset
            self.internal_reduce_global_asset(&self.wnear.clone(), &U128::from(amount))
        }
        Promise::new(receiver_id.clone()).transfer(amount)
    }
```

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.

```
    pub fn create_dca(&mut self, name: String, token_in: AccountId, token_out: AccountId,
         single_amount_in: U128,
                     start_time: u64, interval_time: u64, count: u16, lowest_price: u64,
                         highest_price: u64, slippage: u16) -> bool {
        // record storage fee
        let initial_storage_usage = env::storage_usage();
        let user = env::predecessor_account_id();
        require!(slippage >= MIN_SLIPPAGE, SLIPPAGE_TOO_SMALL);
        require!(start_time > env::block_timestamp_ms(), INVALID_START_TIME);
        require!(self.deposit_limit_map.contains_key(&token_in) && self.deposit_limit_map.
             contains_key(&token_out), INVALID_TOKEN);
        if self.status != DCAStatus::Running {
            self.internal_create_bot_refund_with_near(&user, &token_in, &token_out, env::
                attached_deposit(), PAUSE_OR_SHUTDOWN);
            return false;
        }
        let total_amount_in = single_amount_in.0 * (count as u128);
        if self.internal_get_user_balance(&user, &token_in).0 < total_amount_in {
            self.internal_create_bot_refund_with_near(&user, &token_in, &token_out, env::
                attached_deposit(), LESS_TOKEN_IN);
            return false;
        }
        // create id
        let next_id = self.internal_get_and_use_next_id().to_string();
        let next_dca_key = self.internal_get_dca_key(next_id);
        let dca_vault = DCAVault {
            name,
            user: user.clone(),
            id: next_dca_key.clone(),
            closed: false,
            token_in: token_in.clone(),
            token_out,
            start_time,
            interval_time,
            single_amount_in,
            count,
            execute_count: 0,
            lowest_price,
            highest_price,
            left_amount_in: U128::from(total_amount_in),
            buy_amount_record: U128::from(0),
            slippage,
            process: DCA_STATUS_NORMAL,
            locked: false,
            need_withdraw_amount: U128::from(0),
            buy_amount_to_user: false,
        };
        self.dca_vault_map.insert(&next_dca_key, &dca_vault);
        emit::create_dca(dca_vault);
        // add locked asset
        self.internal_transfer_assets_to_lock(&user, &token_in, U128::from(total_amount_in));


        // refund storage fee
        self.internal_refund_deposit(env::attached_deposit(), initial_storage_usage, &user);
        return true;
    }
```

Listing 2.23: dca.rs

```
pub fn execute_dca(&mut self, vault_id: String, swap_msg: String) {
    require!(env::attached_deposit() == EXECUTE_DCA_FEE);
    require!(self.status == DCAStatus::Running, PAUSE_OR_SHUTDOWN);
    require!(self.market_user_map.contains_key(&(env::predecessor_account_id())), INVALID_USER)
        ;
    require!(self.market_user_map.get(&(env::predecessor_account_id())).unwrap(), INVALID_USER)
        ;
    require!(self.dca_vault_map.contains_key(&vault_id), INVALID_VAULT_ID);
    let mut dca_vault = self.dca_vault_map.get(&vault_id).unwrap();
    require!(!dca_vault.locked, LOCKED);
    require!(!dca_vault.closed, DCA_CLOSED);
    self.internal_check_dca_buy_available(&dca_vault);
    if dca_vault.process == DCA_STATUS_SWAPPED {
        self.internal_ref_withdraw(&mut dca_vault);
```

```
            return;
        }
        let pair_key = self.internal_get_pair_key(&dca_vault.token_in, &dca_vault.token_out);
        let path_op = self.recorded_pair_path.get(&pair_key);
        // execute buy
        if self.internal_check_need_oracle(&dca_vault) {
            // require oracle
            self.get_price_for_execute(&mut dca_vault, swap_msg);
        } else if path_op.is_some() && path_op.clone().unwrap().len() > 0 {
            // check mint amount out
            let single_amount_out = dca_vault.single_amount_in.0;
            self.internal_ref_estimate(&mut dca_vault, swap_msg, path_op.unwrap(), 0,
                single_amount_out);
        } else {
            // direct buy
            self.internal_execute_buy(&mut dca_vault, swap_msg, None, None);
        }
    }
```

Listing 2.24: dca.rs

```
    pub fn internal_execute_buy(&mut self, dca_vault: &mut DCAVault, swap_msg: String,
        price_list_op: Option<Vec<Price>>, estimate_amount_out_op: Option<u128>) -> bool {
        require!(self.status == DCAStatus::Running, PAUSE_OR_SHUTDOWN);
        require!(!dca_vault.locked, LOCKED);
        require!(!dca_vault.closed, DCA_CLOSED);
        require!(self.internal_get_user_locked_balance(&dca_vault.user, &dca_vault.token_in).0 >=
            dca_vault.single_amount_in.0, LESS_TOKEN_IN);
        let swap_info = serde_json::from_str::<SwapMessage>(&swap_msg).expect(INVALID_EXECUTE_MSG);
        // check swap param
        let (amount_in, amount_out) = self.internal_check_swap_info(&swap_info, &dca_vault);
        // check oracle
        self.internal_check_oracle_price(&dca_vault, price_list_op, amount_in, amount_out);
        // check estimate amount out
        self.internal_check_estimate_amount_out(amount_out, estimate_amount_out_op, dca_vault.
            slippage);


        self.internal_lock_dca_vault(dca_vault);
        if dca_vault.process == DCA_STATUS_NORMAL {
            // 1.deposit,2.swap,3.withdraw
            self.internal_ref_deposit(dca_vault, amount_in, &swap_info);
        } else if dca_vault.process == DCA_STATUS_DEPOSITED {
            // 1.swap, 2.withdraw
            self.internal_ref_swap(dca_vault, amount_in, &swap_info);
        }
        // else {
        //     // 1.withdraw
        //     self.internal_ref_withdraw(dca_vault);
        // }
        return true;
    }
```

Listing 2.25: dca_private.rs

```rust
pub fn internal_ref_swap(&mut self, dca_vault: &mut DCAVault, amount_in: u128, swap_info: &
     SwapMessage) {
    // get referral
    let referral = if swap_info.referral_id.is_some() { Some(AccountId::new_unchecked(swap_info
        .referral_id.clone().unwrap().to_string())) } else { None };
    ext_ref::ext(self.ref_exchange_id.clone())
        .with_attached_deposit(1)
        .with_static_gas(GAS_FOR_SWAP)
        .execute_actions(
            swap_info.actions.clone(),
            referral
        ).then(
        Self::ext(env::current_account_id())
            .with_static_gas(GAS_FOR_SWAP_CALL_BACK)
            .callback_do_swap(dca_vault, amount_in)
    );
}
```

Listing 2.26: dca_private.rs

```rust
pub fn callback_do_swap(&mut self, dca_vault: &mut DCAVault, amount_in: u128) {
    if !is_promise_success() {
        emit::ref_swap_failed(&dca_vault.user, amount_in, &dca_vault.token_in, &dca_vault.
             token_out);
        self.internal_unlock_dca_vault(dca_vault);
        return;
    }


    let cross_call_result = promise_result_as_success().expect(ERR102_CROSS_CONTRACT_FAILED);
    let action_result = serde_json::from_slice::<ActionResult>(&cross_call_result).unwrap();
    match action_result {
        ActionResult::None => {
            emit::ref_swap_failed(&dca_vault.user, amount_in, &dca_vault.token_in, &dca_vault.
                 token_out);
            self.internal_unlock_dca_vault(dca_vault);
        }
        ActionResult::Amount(amount) => {
            emit::ref_swap_success(&dca_vault.user, amount_in, amount.0, &dca_vault.token_in, &
                 dca_vault.token_out);
            // record to swapped
            self.internal_record_ref_swapped(dca_vault, amount.clone());
            // withdraw
            self.internal_ref_withdraw(dca_vault);
        }
    }
}
```

Listing 2.27: dca_callback.rs

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

```
            .with_static_gas(GAS_FOR_WITHDRAW)
            .withdraw(
                dca_vault.token_out.clone(),
                dca_vault.need_withdraw_amount.clone(),
                None,
                None
            ).then(
            Self::ext(env::current_account_id())
                .with_static_gas(GAS_FOR_WITHDRAW_CALL_BACK)
                .callback_do_withdraw(dca_vault, dca_vault.single_amount_in.0, dca_vault.
                    need_withdraw_amount.0)
        );
    }
```

Listing 2.28: dca_private.rs

```
    pub fn callback_do_withdraw(&mut self, dca_vault: &mut DCAVault, amount_in: u128, amount_out:
        u128) {
        if !is_promise_success() {
            emit::ref_withdraw_failed(&dca_vault.user, amount_out, &dca_vault.token_out);
            self.internal_unlock_dca_vault(dca_vault);
            return;
        }
        emit::ref_withdraw_succeeded(&dca_vault.user, amount_out, &dca_vault.token_out);
        // complete once dca
        // calculate protocol fee
        let (real_amount_out, protocol_fee) = self.internal_calculate_protocol_fee(amount_out);


        self.internal_record_ref_normal(dca_vault, amount_in, real_amount_out);
        // update asset
        self.internal_increase_locked_assets(&dca_vault.user, &dca_vault.token_out, &U128::from(
            real_amount_out));
        // update global asset
        self.internal_increase_global_asset(&dca_vault.token_out, &U128::from(amount_out));
        // add protocol asset
        self.internal_increase_protocol_fee(&dca_vault.token_out, &U128::from(protocol_fee));
    }
```

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.

```rust
pub fn execute_dca(&mut self, vault_id: String, swap_msg: String) {
    require!(env::attached_deposit() == EXECUTE_DCA_FEE);
    require!(self.status == DCAStatus::Running, PAUSE_OR_SHUTDOWN);
    require!(self.market_user_map.get(&(env::predecessor_account_id())).unwrap(), INVALID_USER)
        ;
    require!(self.dca_vault_map.contains_key(&vault_id), INVALID_VAULT_ID);
    let mut dca_vault = self.dca_vault_map.get(&vault_id).unwrap();
    require!(!dca_vault.locked, LOCKED);
    require!(!dca_vault.closed, DCA_CLOSED);
    self.internal_check_dca_buy_available(&dca_vault);
    if dca_vault.process == DCA_STATUS_SWAPPED {
        self.internal_ref_withdraw(&mut dca_vault);
        return;
    }
    let pair_path_key = self.internal_get_pair_key(&dca_vault.token_in, &dca_vault.token_out);
    let path_op = self.recorded_pair_path.get(&pair_path_key);
    // execute buy
    if self.internal_check_need_oracle(&dca_vault) {
        // require oracle
        self.get_price_for_execute(&mut dca_vault, swap_msg);
    } else if path_op.is_some() && path_op.clone().unwrap().len() > 0 {
        // check mint amount out
        let single_amount_out = dca_vault.single_amount_in.0;
        self.internal_ref_estimate(&mut dca_vault, swap_msg, path_op.unwrap(), 0,
            single_amount_out);
    } else {
        // direct buy
        self.internal_execute_buy(&mut dca_vault, swap_msg, None, None);
    }
}
```

Listing 2.30: dca.rs

```rust
pub fn internal_ref_withdraw(&mut self, dca_vault: &mut DCAVault) {
    ext_ref::ext(self.ref_exchange_id.clone())
        .with_attached_deposit(1)
        .with_static_gas(GAS_FOR_WITHDRAW)
        .withdraw(
            dca_vault.token_out.clone(),
            dca_vault.need_withdraw_amount.clone(),
            None,
            None
        ).then(
        Self::ext(env::current_account_id())
            .with_static_gas(GAS_FOR_WITHDRAW_CALL_BACK)
            .callback_do_withdraw(dca_vault, dca_vault.single_amount_in.0, dca_vault.need_withdraw_amount.0)
    );
}
```

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);
```

```rust
    let dca_vault = DCAVault {
        name,
        user: user.clone(),
        id: next_dca_key.clone(),
        closed: false,
        token_in: token_in.clone(),
        token_out,
        start_time,
        interval_time,
        single_amount_in,
        count,
        execute_count: 0,
        lowest_price,
        highest_price,
        left_amount_in: U128::from(total_amount_in),
        buy_amount_record: U128::from(0),
        slippage,
        process: DCA_STATUS_NORMAL,
        locked: false,
        need_withdraw_amount: U128::from(0),
        buy_amount_to_user: false,
    };
    self.dca_vault_map.insert(&next_dca_key, &dca_vault);
    emit::create_dca(dca_vault);
    // add locked asset
    self.internal_transfer_assets_to_lock(&user, &token_in, U128::from(total_amount_in));


    // refund storage fee
    self.internal_refund_deposit(env::attached_deposit(), initial_storage_usage, &user);
    return true;
}
```

Listing 2.32: dca.rs

```rust
pub fn internal_reduce_asset(&mut self, user: &AccountId, token: &AccountId, amount: &U256C) {
    let mut user_balances = self.user_balances_map.get(user).unwrap_or_else(|| {
        let mut map = LookupMap::new(StorageKey::UserBalanceSubKey(user.clone()));
        map.insert(token, &U256C::from(0));
        map
    });


    let balance = user_balances.get(token).unwrap_or(U256C::from(0));
    user_balances.insert(token, &(balance - amount));


    self.user_balances_map.insert(user, &user_balances);
}
```

Listing 2.33: grid_bot_asset.rs

```rust
pub fn set_per_grid_storage_fee(&mut self, new_per_grid_storage_fee: U128) {
        self.assert_owner();
        self.per_grid_storage_fee = new_per_grid_storage_fee.0;
    }
```

Listing 2.34: dca_owner.rs

```rust
    pub fn execute_dca(&mut self, vault_id: String, swap_msg: String) {
        require!(env::attached_deposit() == EXECUTE_DCA_FEE);
        require!(self.status == DCAStatus::Running, PAUSE_OR_SHUTDOWN);
        require!(self.market_user_map.contains_key(&(env::predecessor_account_id())), INVALID_USER)
            ;
        require!(self.market_user_map.get(&(env::predecessor_account_id())).unwrap(), INVALID_USER)
            ;
        require!(self.dca_vault_map.contains_key(&vault_id), INVALID_VAULT_ID);
        let mut dca_vault = self.dca_vault_map.get(&vault_id).unwrap();
        require!(!dca_vault.locked, LOCKED);
        require!(!dca_vault.closed, DCA_CLOSED);
        self.internal_check_dca_buy_available(&dca_vault);
        if dca_vault.process == DCA_STATUS_SWAPPED {
            self.internal_ref_withdraw(&mut dca_vault);
            return;
        }
        let pair_key = self.internal_get_pair_key(&dca_vault.token_in, &dca_vault.token_out);
        let path_op = self.recorded_pair_path.get(&pair_key);
        // execute buy
        if self.internal_check_need_oracle(&dca_vault) {
            // require oracle
            self.get_price_for_execute(&mut dca_vault, swap_msg);
        } else if path_op.is_some() && path_op.clone().unwrap().len() > 0 {
            // check mint amount out
            let single_amount_out = dca_vault.single_amount_in.0;
            self.internal_ref_estimate(&mut dca_vault, swap_msg, path_op.unwrap(), 0,
                single_amount_out);
        } else {
            // direct buy
            self.internal_execute_buy(&mut dca_vault, swap_msg, None, None);
        }
    }
```

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.

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

Listing 2.36: dca.rs

```rust
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>) {
    require!(env::predecessor_account_id() == self.owner_id, ERR_NOT_ALLOWED);
    require!(env::attached_deposit() == REGISTER_PAIR_STORAGE_FEE * 2, LESS_STORAGE_FEE);
    require!(token_a == path.get(0).unwrap().token_in && token_b == path.get(path.len() - 1).
        unwrap().token_out, INVALID_TOKEN);


    let pair_key = self.internal_get_pair_key(&token_a, &token_b);
    // record Pair pool id
    self.recorded_pair_path.insert(&pair_key, &path);


    self.deposit_limit_map.insert(&token_a, &token_a_min_deposit);
    self.deposit_limit_map.insert(&token_b, &token_b_min_deposit);


    self.internal_increase_global_asset(&token_a, &U128::from(0));
    self.internal_increase_global_asset(&token_b, &U128::from(0));


    self.internal_increase_protocol_fee(&token_a, &U128::from(0));
    self.internal_increase_protocol_fee(&token_b, &U128::from(0));


    self.internal_increase_locked_in_ref_asset(&token_a, &U128::from(0));
    self.internal_increase_locked_in_ref_asset(&token_b, &U128::from(0));


    self.internal_set_oracle(&token_a, token_a_oracle_id_op);
    self.internal_set_oracle(&token_b, token_b_oracle_id_op);


    self.internal_storage_deposit(&env::current_account_id(), &token_a,
        REGISTER_TOKEN_STORAGE_FEE);
    self.internal_storage_deposit(&env::current_account_id(), &token_b, REGISTER_TOKEN_STORAGE_FEE);


    self.internal_ref_storage_deposit(&env::current_account_id(), REGISTER_TOKEN_STORAGE_FEE);
}


#[payable]
pub fn set_min_deposit(&mut self, token: AccountId, min_deposit: U128) {
    require!(env::predecessor_account_id() == self.owner_id, ERR_NOT_ALLOWED);
    require!(env::attached_deposit() == DEFAULT_CONFIG_SET_STORAGE_FEE, LESS_STORAGE_FEE);
    self.deposit_limit_map.insert(&token, &min_deposit);
}


#[payable]
pub fn storage_deposit(&mut self, token: AccountId, storage_fee: U128) {
    require!(env::predecessor_account_id() == self.owner_id, ERR_NOT_ALLOWED);
    require!(env::attached_deposit() == storage_fee.0, LESS_TOKEN_STORAGE_FEE);
    self.internal_storage_deposit(&env::current_account_id(), &token, storage_fee.0);
}


#[payable]
pub fn enable_oracle_config(&mut self, token: AccountId, oracle_id: String) {
    require!(env::predecessor_account_id() == self.owner_id, ERR_NOT_ALLOWED);
    require!(env::attached_deposit() == DEFAULT_CONFIG_SET_STORAGE_FEE, LESS_STORAGE_FEE);
    self.internal_set_oracle(&token, Some(oracle_id));
}
```

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.

```rust
    pub fn withdraw_near(&mut self, user: &AccountId, amount: u128) {
        ext_wnear::ext(self.wnear.clone())
            .with_attached_deposit(ONE_YOCTO)
            .near_withdraw(U128::from(amount))
            .then(
                Self::ext(env::current_account_id())
                    .after_withdraw_near(
                        user,
                        amount,
                )
        );
}
```

Listing 2.38: wnear.rs

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

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.

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

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.

```rust
pub fn internal_ref_estimate(&mut self, dca_vault: &mut DCAVault, swap_msg: String, path: Vec<
    Pool>, current_path: u8, amount_in: u128) -> Promise {
    let pool = path.get(current_path as usize).unwrap();
    ext_ref::ext(self.ref_exchange_id.clone())
        .with_static_gas(GAS_FOR_REF_ESTIMATE_ONCE)
        .get_return(pool.pool_id.clone(), pool.token_in.clone(), U128::from(amount_in), pool.
            token_out.clone())
        .then(
        Self::ext(env::current_account_id())
            .with_static_gas(Gas(GAS_FOR_REF_ESTIMATE.0 - GAS_FOR_REF_ESTIMATE_ONCE.0 * (
                current_path as u64)))
            .callback_ref_estimate(
                dca_vault,
                swap_msg,
                path,
                current_path,
                amount_in,
            )
    )
}
```

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.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS