



BlockSec

Security Audit Report for Ref-Exchange

Date: Nov 2, 2022

Version: 1.0

Contact: contact@blocksec.com

Contents

1	Introduction	1
1.1	About Target Contracts	1
1.2	Disclaimer	2
1.3	Procedure of Auditing	2
1.3.1	Software Security	2
1.3.2	DeFi Security	3
1.3.3	NFT Security	3
1.3.4	Additional Recommendation	3
1.4	Security Model	3
2	Findings	5
2.1	Software Security	5
2.1.1	Improper Account Unregistration	5
2.2	DeFi Security	6
2.2.1	Unrestricted Referral Account	6
2.2.2	Incorrect Admin Fees Calculation in Simple Pool	7
2.3	Additional Recommendation	9
2.3.1	Lack of Check on Guardians' Removal	9
2.3.2	Two-Step Transfer of Privileged Account Ownership	10
2.3.3	Potential Elastic Supply Token Problem	10
2.3.4	Improper Check on the Admin Fees	10
2.3.5	Lack of Check in retrieve_unmanaged_token()	11
2.3.6	Lack of Check on the Gas Used by migrate()	12
2.3.7	Code Optimization (I)	13
2.3.8	Code Optimization (II)	16
2.4	Notes	17
2.4.1	Delayed Price in Rated Swap Pool	17
2.4.2	Timely Triggering update_token_rate()	18
2.4.3	Sensitive Functions Managed by DAO	18

Report Manifest

Item	Description
Client	Ref Finance
Target	Ref-Exchange

Version History

Version	Date	Description
1.0	November 2, 2022	First Release

About BlockSec The **BlockSec Team** focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team 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 released detailed analysis reports of high-impact security incidents. 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 repository that has been audited includes ref-exchange ¹.

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

Project		Commit SHA
Ref-Exchange	Version 1	536a60c842e018a535b478c874c747bde82390dd

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

```
1 ref-exchange
2 |-- src
3 | |-- rated_swap
4 | | |-- linear_rate.rs
5 | | |-- math.rs
6 | | |-- mod.rs
7 | | |-- nearx_rate.rs
8 | | |-- rate.rs
9 | | |-- stnear.rs
10 | |
11 | |-- stable_swap
12 | | |-- math.rs
13 | | |-- mod.rs
14 | |
15 | |-- account_deposit.rs
16 | |-- action.rs
17 | |-- admin_fee.rs
18 | |-- custom_keys.rs
19 | |-- errors.rs
20 | |-- legacy.rs
21 | |-- lib.rs
22 | |-- multi_fungible_token.rs
23 | |-- owner.rs
24 | |-- pool.rs
25 | |-- simple_pool.rs
```

¹<https://github.com/ref-finance/ref-contracts/tree/main/ref-exchange>

```
26 | |-- storage_impl.rs
27 | |-- token_receiver.rs
28 | |-- utils.rs
29 | |-- views.rs
```

Listing 1.1: Audit Scope for this Report

1.2 Disclaimer

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

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

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

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

- * Reentrancy
- * DoS
- * Access control
- * Data handling and data flow
- * Exception handling

- * Untrusted external call and control flow
- * Initialization consistency
- * Events operation
- * Error-prone randomness
- * Improper use of the proxy system

1.3.2 DeFi Security

- * Semantic consistency
- * Functionality consistency
- * Access control
- * 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.

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

³<https://cwe.mitre.org/>

Table 1.1: Vulnerability Severity Classification

Impact	High	High	Medium
	Low	Medium	Low
		High	Low
		Likelihood	

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.

Chapter 2 Findings

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

- High Risk: 0
- Medium Risk: 2
- Low Risk: 1
- Recommendations: 8
- Notes: 3

ID	Severity	Description	Category	Status
1	Medium	Improper Account Unregistration	Software Security	Confirmed
2	Low	Unrestricted Referral Account	DeFi Security	Confirmed
3	Medium	Incorrect Admin Fees Calculation in Simple Pool	DeFi Security	Confirmed
4	-	Lack of Check on Guardians' Removal	Recommendation	Confirmed
5	-	Two-Step Transfer of Privileged Account Ownership	Recommendation	Confirmed
6	-	Potential Elastic Supply Token Problem	Recommendation	Confirmed
7	-	Improper Check on the Admin Fees	Recommendation	Confirmed
8	-	Lack of Check in <code>retrieve_unmanaged_token()</code>	Recommendation	Confirmed
9	-	Lack of Check on the Gas Used by <code>migrate()</code>	Recommendation	Confirmed
10	-	Code Optimization (I)	Recommendation	Confirmed
11	-	Code Optimization (II)	Recommendation	Confirmed
12	-	Delayed Price in Rated Swap Pool	Note	Confirmed
13	-	Timely Triggering <code>update_token_rate()</code>	Note	Confirmed
14	-	Sensitive Functions Managed by DAO	Note	Confirmed

The details are provided in the following sections.

2.1 Software Security

2.1.1 Improper Account Unregistration

Severity Medium

Status Confirmed

Introduced by [Version 1](#)

Description Function `storage_unregister()` allows users to unregister their accounts, and get back their deposits (i.e., [NEARs](#)). However, it doesn't check whether the `legacy_tokens` of accounts are empty before the unregistration.

```
56  #[allow(unused_variables)]
57  #[payable]
58  fn storage_unregister(&mut self, force: Option<bool>) -> bool {
59      assert_one_yocto();
60      self.assert_contract_running();
61      let account_id = env::predecessor_account_id();
```



```
62     if let Some(account_deposit) = self.internal_get_account(&account_id) {
63         // TODO: figure out force option logic.
64         assert!(
65             account_deposit.tokens.is_empty(),
66             "{}", ERR18_TOKENS_NOT_EMPTY
67         );
68         self.accounts.remove(&account_id);
69         Promise::new(account_id.clone()).transfer(account_deposit.near_amount);
70         true
71     } else {
72         false
73     }
74 }
```

Listing 2.1: src/storage_impl.rs

Impact Users may lose tokens that are recorded in `legacy_tokens`.

Suggestion I Add the check to ensure `legacy_tokens` of accounts are empty before the removal.

Feedback from the Project Will fix this in the next accumulated contract upgrade.

2.2 DeFi Security

2.2.1 Unrestricted Referral Account

Severity Low

Status Confirmed

Introduced by Version 1

Description The protocol allows the user to provide a `referral` account for receiving a reward during the swap process. However, there is no restriction on this `referral` account, which allows the user to receive the `referral` fee as a reward by providing his/her own address.

```
62     #[allow(unreachable_code)]
63     fn ft_on_transfer(
64         &mut self,
65         sender_id: ValidAccountId,
66         amount: U128,
67         msg: String,
68     ) -> PromiseOrValue<U128> {
69         self.assert_contract_running();
70         let token_in = env::predecessor_account_id();
71         // feature frozenlist
72         self.assert_no_frozen_tokens(&[token_in.clone()]);
73         if msg.is_empty() {
74             // Simple deposit.
75             self.internal_deposit(sender_id.as_ref(), &token_in, amount.into());
76             PromiseOrValue::Value(U128(0))
77         } else {
78             // instant swap
79             let message =
80                 serde_json::from_str:::<TokenReceiverMessage>(&msg).expect(ERR28_WRONG_MSG_FORMAT);
```

```

81         match message {
82             TokenReceiverMessage::Execute {
83                 referral_id,
84                 actions,
85             } => {
86                 let referral_id = referral_id.map(|x| x.to_string());
87                 let out_amounts = self.internal_direct_actions(
88                     token_in,
89                     amount.0,
90                     referral_id,
91                     &actions,
92                 );
93                 for (token_out, amount_out) in out_amounts.into_iter() {
94                     self.internal_send_tokens(sender_id.as_ref(), &token_out, amount_out);
95                 }
96                 // Even if send tokens fails, we don't return funds back to sender.
97                 PromiseOrValue::Value(U128(0))
98             }
99         }
100     }
101 }

```

Listing 2.2: src/token_receiver.rs

Impact Users can earn the additional referral fee in the swap process, which is against the original design.

Suggestion I Ensure the `referral` account is different from the `sender_id`.

Feedback from the Project The new rated referral fee feature would include a fix for it.

2.2.2 Incorrect Admin Fees Calculation in Simple Pool

Severity Medium

Status Confirmed

Introduced by Version 1

Description The shares minted to admin for Simple Pool are calculated as follows:

$$Minted_Share = Total_Share * Admin_Fee * \frac{\sqrt{k'} - \sqrt{k}}{\sqrt{k'}}$$

The actual `Admin_Fee` is $\frac{Admin_Fee_Amount}{Total_Fee_Amount}$. The total value of the pool can be represented as $\sqrt{k'}$, and the `Total_Fee_Amount` can be represented as $\sqrt{k'} - \sqrt{k}$. Thus, `Admin_Fee_Amount` should be

$$\frac{Minted_Share}{Total_Share + Minted_Share} * \sqrt{k'}$$

In this case, given the `Minted_Share` above, the actual `Admin_Fee` could be calculated as follows:

$$Actual_Admin_Fee = \frac{Total_Share * Admin_Fee * \frac{\sqrt{k'} - \sqrt{k}}{\sqrt{k'}}}{Total_Share * Admin_Fee * \frac{\sqrt{k'} - \sqrt{k}}{\sqrt{k'}} + Total_Share} * \frac{\sqrt{k'}}{\sqrt{k'} - \sqrt{k}} = \frac{1}{\frac{1}{Admin_Fee} + \frac{\sqrt{k'} - \sqrt{k}}{\sqrt{k'}}}$$

, which is always less than the `Admin_Fee` in the `Simple Pool`. That's to say, the calculation of the amount of shares minted for the admin is incorrect.

To ensure that the actual $Admin_Fee$ is equal to $Admin_Fee$, we have the following equations:

$$ActualAdmin_Fee = \frac{Admin_Fee_Amount}{Total_Fee_Amount} = \frac{Minted_Share * \sqrt{k'}}{Total_Share + Minted_Share} \div (\sqrt{k'} - \sqrt{k}) = Admin_Fee$$

Given the formula above, the minted share should be calculated as following:

$$Minted_Share = Total_Share * \frac{\sqrt{k'} - \sqrt{k}}{(\frac{1}{Admin_Fee} - 1) * \sqrt{k'} + \sqrt{k}}$$

```

269 pub fn swap(
270     &mut self,
271     token_in: &AccountId,
272     amount_in: Balance,
273     token_out: &AccountId,
274     min_amount_out: Balance,
275     admin_fee: &AdminFees,
276 ) -> Balance {
277     assert_ne!(token_in, token_out, "{}", ERR73_SAME_TOKEN);
278     let in_idx = self.token_index(token_in);
279     let out_idx = self.token_index(token_out);
280     let amount_out = self.internal_get_return(in_idx, amount_in, out_idx);
281     assert!(amount_out >= min_amount_out, "{}", ERR68_SLIPPAGE);
282     env::log(
283         format!(
284             "Swapped {} {} for {} {}",
285             amount_in, token_in, amount_out, token_out
286         )
287         .as_bytes(),
288     );
289
290     let prev_invariant =
291         integer_sqrt(U256::from(self.amounts[in_idx]) * U256::from(self.amounts[out_idx]));
292
293     self.amounts[in_idx] += amount_in;
294     self.amounts[out_idx] -= amount_out;
295
296     // "Invariant" is by how much the dot product of amounts increased due to fees.
297     let new_invariant =
298         integer_sqrt(U256::from(self.amounts[in_idx]) * U256::from(self.amounts[out_idx]));
299
300     // Invariant can not reduce (otherwise loosing balance of the pool and something it broken)
301
302     assert!(new_invariant >= prev_invariant, "{}", ERR75_INVARIANT_REDUCE);
303     let numerator = (new_invariant - prev_invariant) * U256::from(self.shares_total_supply);
304
305     // Allocate exchange fee as fraction of total fee by issuing LP shares proportionally.
306     if admin_fee.exchange_fee > 0 && numerator > U256::zero() {
307         let denominator = new_invariant * FEE_DIVISOR / admin_fee.exchange_fee;
308         self.mint_shares(&admin_fee.exchange_id, (numerator / denominator).as_u128());
309     }
310
311     // If there is referral provided and the account already registered LP, allocate it % of LP
312     rewards.

```

```
311     if let Some(referral_id) = &admin_fee.referral_id {
312         if admin_fee.referral_fee > 0
313             && numerator > U256::zero()
314             && self.shares.contains_key(referral_id)
315         {
316             let denominator = new_invariant * FEE_DIVISOR / admin_fee.referral_fee;
317             self.mint_shares(referral_id, (numerator / denominator).as_u128());
318         }
319     }
320
321     // Keeping track of volume per each input traded separately.
322     // Reported volume with fees will be sum of input, without fees will be sum of output.
323     self.volumes[in_idx].input.0 += amount_in;
324     self.volumes[in_idx].output.0 += amount_out;
325
326     amount_out
327 }
```

Listing 2.3: src/simple_pool.rs

Impact Simple Pool will always charge less admin fees than expected.

Suggestion I Use the equation listed above to calculate the shares minted for admins.

Feedback from the Project Will fix this in the next accumulated contract upgrade

2.3 Additional Recommendation

2.3.1 Lack of Check on Guardians' Removal

Status Confirmed

Introduced by Version 1

Description The `owner` of the protocol can remove `guardians` via the function `remove_guardians()`. However, the existence of `guardians` is not checked. In this case, if the `guardians` do not exist, the program will not panic, which may mislead the `owner` and bring unexpected impact.

```
64     #[payable]
65     pub fn remove_guardians(&mut self, guardians: Vec<ValidAccountId>) {
66         assert_one_yocto();
67         self.assert_owner();
68         for guardian in guardians {
69             self.guardians.remove(guardian.as_ref());
70         }
71     }
```

Listing 2.4: src/owner.rs

Suggestion I Check the return value of function `remove_guardians()`.

Feedback from the Project Will fix it in the next accumulated contract upgrade.

2.3.2 Two-Step Transfer of Privileged Account Ownership

Status Confirmed

Introduced by [Version 1](#)

Description The contract uses `set_owner()` to configure the privileged account, which can conduct many sensitive operations (e.g., retrieve unmanaged tokens). In this case, when an incorrect new owner is provided, the contract is under the risk of attack and the privileged functions cannot be invoked.

```
14  #[payable]
15  pub fn set_owner(&mut self, owner_id: ValidAccountId) {
16      assert_one_yocto();
17      self.assert_owner();
18      self.owner_id = owner_id.as_ref().clone();
19  }
```

Listing 2.5: `src/owner.rs`

Suggestion I Implement a two-step approach for the `owner` update: `set_owner()` and `commit_owner()`.

Feedback from the Project To prevent human unintentional errors during the ownership transfer, we would have a safety design to ensure the next owner exists and is able to perform his duty (sign TX). For that purpose, we may leverage a relay baton process: Grant (by cur owner with a deadline blockheight or timestamp), Accept (by next owner to ensure he can sign TX within the deadline), Confirm (by cur owner and followed by the real ownership transfer) or Cancel (by cur owner)

2.3.3 Potential Elastic Supply Token Problem

Status Confirmed

Introduced by [Version 1](#)

Description Elastic supply tokens could dynamically adjust their price, supply, user's balance, etc. For example, inflation tokens, deflation tokens, rebasing tokens, and so forth. In the current implementation of protocol, elastic supply tokens are not supported. If the token is a deflation token, there will be a difference between the recorded amount of transferred tokens to this smart contract (as a parameter of function `ft_on_transfer()`) and the actual number of transferred tokens (the token smart contract itself). That's because a small number of tokens will be burned by the token smart contract.

This inconsistency can lead to security impacts for the operations based on the transferred amount of tokens.

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

2.3.4 Improper Check on the Admin Fees

Status Confirmed

Introduced by [Version 1](#)

Description In the process of swapping, the user has to pay three different fees for the service, i.e., `exchange_fee`, `referral_fee`, and `"lp_fee"`. The admin fee (i.e., `exchange_fee` and `referral_fee`) is adjusted with the function `modify_admin_fee()`. However, the maximum admin fee (the sum of `exchange_fee`

and `referral_fee`) is allowed to be set as `FEE_DIVISOR` (i.e., 100%), which means all the fees collected from the user are kept to admin. In this case, the liquidity provider cannot get any profit, which is unfair.

```
137  #[payable]
138  pub fn modify_admin_fee(&mut self, exchange_fee: u32, referral_fee: u32) {
139      assert_one_yocto();
140      self.assert_owner();
141      assert!(exchange_fee + referral_fee <= FEE_DIVISOR, "{}", ERR101_ILLEGAL_FEE);
142      self.exchange_fee = exchange_fee;
143      self.referral_fee = referral_fee;
144  }
```

Listing 2.6: `src/owner.rs`

Suggestion I It is recommended to limit the sum up of `exchange_fee` + `referral_fee` with a reasonable value, which is less than `FEE_DIVISOR`.

Feedback from the Project Will fix it in the next accumulated contract upgrade.

2.3.5 Lack of Check in `retrieve_unmanaged_token()`

Status Confirmed

Introduced by Version 1

Description Function `retrieve_unmanaged_token()` enables the `owner` to transfer `NEP-141` tokens from the contract to the `owner`. The purpose is to retrieve the tokens accidentally transferred in by others. However, there is no limitation on the amount of tokens that are transferred out. In this case, users' assets may lose if the owner transfers more tokens than expected.

```
29  #[payable]
30  pub fn retrieve_unmanaged_token(&mut self, token_id: ValidAccountId, amount: U128) -> Promise
31  {
32      self.assert_owner();
33      assert_one_yocto();
34      let token_id: AccountId = token_id.into();
35      let amount: u128 = amount.into();
36      assert!(amount > 0, "{}", ERR29_ILLEGAL_WITHDRAW_AMOUNT);
37      env::log(
38          format!(
39              "Going to retrieve token {} to owner, amount: {}",
40              &token_id, amount
41          )
42      );
43      ext_fungible_token::ft_transfer(
44          self.owner_id.clone(),
45          U128(amount),
46          None,
47          &token_id,
48          1,
49          env::prepaid_gas() - GAS_FOR_BASIC_OP,
50      )
51  }
```

Listing 2.7: src/owner.rs

Suggestion I It is recommended to add the check to ensure the user's assets would not be transferred out.

Feedback from the Project Current safety policy includes two points: First, we only grant that sensitive interface to the owner's management, and he (the DAO) would be careful with the numbers according to relevant transfer TX. Second, this interface can only withdraw tokens to the owner's account, which gives the owner (the DAO) 2ed chance to inspect numbers.

2.3.6 Lack of Check on the Gas Used by migrate()

Status Confirmed

Introduced by Version 1

Description There is no check on whether the `attached_gas` is enough for function `migrate()`.

```

309  #[no_mangle]
310  pub extern "C" fn upgrade() {
311      env::setup_panic_hook();
312      env::set_blockchain_interface(Box::new(near_blockchain::NearBlockchain {}));
313      let contract: Contract = env::state_read().expect(ERR103_NOT_INITIALIZED);
314      contract.assert_owner();
315      let current_id = env::current_account_id().into_bytes();
316      let method_name = "migrate".as_bytes().to_vec();
317      unsafe {
318          BLOCKCHAIN_INTERFACE.with(|b| {
319              // Load input into register 0.
320              b.borrow()
321                  .as_ref()
322                  .expect(BLOCKCHAIN_INTERFACE_NOT_SET_ERR)
323                  .input(0);
324              let promise_id = b
325                  .borrow()
326                  .as_ref()
327                  .expect(BLOCKCHAIN_INTERFACE_NOT_SET_ERR)
328                  .promise_batch_create(current_id.len() as _, current_id.as_ptr() as _);
329              b.borrow()
330                  .as_ref()
331                  .expect(BLOCKCHAIN_INTERFACE_NOT_SET_ERR)
332                  .promise_batch_action_deploy_contract(promise_id, u64::MAX as _, 0);
333              let attached_gas = env::prepaid_gas() - env::used_gas() - GAS_FOR_MIGRATE_CALL;
334              b.borrow()
335                  .as_ref()
336                  .expect(BLOCKCHAIN_INTERFACE_NOT_SET_ERR)
337                  .promise_batch_action_function_call(
338                      promise_id,
339                      method_name.len() as _,
340                      method_name.as_ptr() as _,
341                      0 as _,
342                      0 as _,

```

```
343         0 as _,
344         attached_gas,
345     );
346 });
347 }
348 }
```

Listing 2.8: src/owner.rs

Suggestion I Check whether the `attached_gas` is larger than a specified value.

Feedback from the Project Will fix it in the next accumulated contract upgrade.

2.3.7 Code Optimization (I)

Status Confirmed

Introduced by Version 1

Description Function `internal_unwrap_or_default_account()` is used to get the stored `Account` in the contract with the `AccountId`. If the `AccountId` is not registered, the function will return a default `Account`. This function is improperly used in the functions listed below (i.e., `add_liquidity()`, `add_stable_liquidity()`, `remove_liquidity()`, and `remove_liquidity_by_tokens()`). Take the function `add_liquidity()` as an example, if the `Account` of the sender doesn't exist (line 266), the withdrawal of the deposited tokens in the newly created `Account` (lines 269 - 271) will always fail.

```
237 #[payable]
238 pub fn add_liquidity(
239     &mut self,
240     pool_id: u64,
241     amounts: Vec<U128>,
242     min_amounts: Option<Vec<U128>>,
243 ) -> U128 {
244     self.assert_contract_running();
245     assert!(
246         env::attached_deposit() > 0,
247         "{}", ERR35_AT_LEAST_ONE_YOCTO
248     );
249     let prev_storage = env::storage_usage();
250     let sender_id = env::predecessor_account_id();
251     let mut amounts: Vec<u128> = amounts.into_iter().map(|amount| amount.into()).collect();
252     let mut pool = self.pools.get(pool_id).expect(ERR85_NO_POOL);
253     // feature frozenlist
254     self.assert_no_frozen_tokens(pool.tokens());
255     // Add amounts given to liquidity first. It will return the balanced amounts.
256     let shares = pool.add_liquidity(
257         &sender_id,
258         &mut amounts,
259     );
260     if let Some(min_amounts) = min_amounts {
261         // Check that all amounts are above request min amounts in case of front running that
262         // changes the exchange rate.
263         for (amount, min_amount) in amounts.iter().zip(min_amounts.iter()) {
264             assert!(amount >= &min_amount.0, "{}", ERR86_MIN_AMOUNT);
265         }
266     }
```



```
264     }
265 }
266 let mut deposits = self.internal_unwrap_or_default_account(&sender_id);
267 let tokens = pool.tokens();
268 // Subtract updated amounts from deposits. This will fail if there is not enough funds for
    any of the tokens.
269 for i in 0..tokens.len() {
270     deposits.withdraw(&tokens[i], amounts[i]);
271 }
272 self.internal_save_account(&sender_id, deposits);
273 self.pools.replace(pool_id, &pool);
274 self.internal_check_storage(prev_storage);
275
276 U128(shares)
277 }
```

Listing 2.9: src/lib.rs

```
284 #[payable]
285 pub fn add_stable_liquidity(
286     &mut self,
287     pool_id: u64,
288     amounts: Vec<U128>,
289     min_shares: U128,
290 ) -> U128 {
291     self.assert_contract_running();
292     assert!(
293         env::attached_deposit() > 0,
294         "{}".format(ERR35_AT_LEAST_ONE_YOCTO)
295     );
296     let prev_storage = env::storage_usage();
297     let sender_id = env::predecessor_account_id();
298     let amounts: Vec<u128> = amounts.into_iter().map(|amount| amount.into()).collect();
299     let mut pool = self.pools.get(pool_id).expect(ERR85_NO_POOL);
300     // feature frozenlist
301     self.assert_no_frozen_tokens(pool.tokens());
302     // Add amounts given to liquidity first. It will return the balanced amounts.
303     let mint_shares = pool.add_stable_liquidity(
304         &sender_id,
305         &amounts,
306         min_shares.into(),
307         AdminFees::new(self.exchange_fee),
308     );
309     let mut deposits = self.internal_unwrap_or_default_account(&sender_id);
310     let tokens = pool.tokens();
311     // Subtract amounts from deposits. This will fail if there is not enough funds for any of
    the tokens.
312     for i in 0..tokens.len() {
313         deposits.withdraw(&tokens[i], amounts[i]);
314     }
315     self.internal_save_account(&sender_id, deposits);
316     self.pools.replace(pool_id, &pool);
317     self.internal_check_storage(prev_storage);
```

```
318
319     mint_shares.into()
320 }
```

Listing 2.10: src/lib.rs

```
333  #[payable]
334  pub fn remove_liquidity(&mut self, pool_id: u64, shares: U128, min_amounts: Vec<U128>) -> Vec<
    U128> {
335      assert_one_yocto();
336      self.assert_contract_running();
337      let prev_storage = env::storage_usage();
338      let sender_id = env::predecessor_account_id();
339      let mut pool = self.pools.get(pool_id).expect(ERR85_NO_POOL);
340      // feature frozenlist
341      self.assert_no_frozen_tokens(pool.tokens());
342      let amounts = pool.remove_liquidity(
343          &sender_id,
344          shares.into(),
345          min_amounts
346              .into_iter()
347              .map(|amount| amount.into())
348              .collect(),
349      );
350      self.pools.replace(pool_id, &pool);
351      let tokens = pool.tokens();
352      let mut deposits = self.internal_unwrap_or_default_account(&sender_id);
353      for i in 0..tokens.len() {
354          deposits.deposit(&tokens[i], amounts[i]);
355      }
356      // Freed up storage balance from LP tokens will be returned to near_balance.
357      if prev_storage > env::storage_usage() {
358          deposits.near_amount +=
359              (prev_storage - env::storage_usage()) as Balance * env::storage_byte_cost();
360      }
361      self.internal_save_account(&sender_id, deposits);
362
363      amounts
364          .into_iter()
365          .map(|amount| amount.into())
366          .collect()
367  }
```

Listing 2.11: src/lib.rs

```
373  #[payable]
374  pub fn remove_liquidity_by_tokens(
375      &mut self, pool_id: u64,
376      amounts: Vec<U128>,
377      max_burn_shares: U128
378  ) -> U128 {
379      assert_one_yocto();
380      self.assert_contract_running();
```

```
381     let prev_storage = env::storage_usage();
382     let sender_id = env::predecessor_account_id();
383     let mut pool = self.pools.get(pool_id).expect(ERR85_NO_POOL);
384     // feature frozenlist
385     self.assert_no_frozen_tokens(pool.tokens());
386     let burn_shares = pool.remove_liquidity_by_tokens(
387         &sender_id,
388         amounts
389             .clone()
390             .into_iter()
391             .map(|amount| amount.into())
392             .collect(),
393         max_burn_shares.into(),
394         AdminFees::new(self.exchange_fee),
395     );
396     self.pools.replace(pool_id, &pool);
397     let tokens = pool.tokens();
398     let mut deposits = self.internal_unwrap_or_default_account(&sender_id);
399     for i in 0..tokens.len() {
400         deposits.deposit(&tokens[i], amounts[i].into());
401     }
402     // Freed up storage balance from LP tokens will be returned to near_balance.
403     if prev_storage > env::storage_usage() {
404         deposits.near_amount +=
405             (prev_storage - env::storage_usage()) as Balance * env::storage_byte_cost();
406     }
407     self.internal_save_account(&sender_id, deposits);
408
409     burn_shares.into()
410 }
```

Listing 2.12: src/lib.rs

Suggestion I Replace the function `internal_unwrap_or_default_account()` with the function `internal_unwrap_account()` in above functions.

Feedback from the Project Will fix it in the next accumulated contract upgrade.

2.3.8 Code Optimization (II)

Status Confirmed

Introduced by Version 1

Description Function `ft_on_transfer()` is a callback function which is used to receive tokens. It will check whether the token transferred in is frozen for both the operation of `deposit` and the operation of `swap`. There also exist checks in the operation of `swap` to make sure the token swapped out is not frozen as well. However, this check will be done for each `swap`. The problem comes when a sequence of swaps executes, and there is a frozen token in the middle of the sequence. In this case, the execution will not fail until it reaches the middle, and the gas is wasted for executing the previous swap actions.

```
62     #[allow(unreachable_code)]
63     fn ft_on_transfer(
```

```
64     &mut self,
65     sender_id: ValidAccountId,
66     amount: U128,
67     msg: String,
68 ) -> PromiseOrValue<U128> {
69     self.assert_contract_running();
70     let token_in = env::predecessor_account_id();
71     // feature frozenlist
72     self.assert_no_frozen_tokens(&[token_in.clone()]);
73     if msg.is_empty() {
74         // Simple deposit.
75         self.internal_deposit(sender_id.as_ref(), &token_in, amount.into());
76         PromiseOrValue::Value(U128(0))
77     } else {
78         // instant swap
79         let message =
80             serde_json::from_str::<TokenReceiverMessage>(&msg).expect(ERR28_WRONG_MSG_FORMAT);
81         match message {
82             TokenReceiverMessage::Execute {
83                 referral_id,
84                 actions,
85             } => {
86                 let referral_id = referral_id.map(|x| x.to_string());
87                 let out_amounts = self.internal_direct_actions(
88                     token_in,
89                     amount.0,
90                     referral_id,
91                     &actions,
92                 );
93                 for (token_out, amount_out) in out_amounts.into_iter() {
94                     self.internal_send_tokens(sender_id.as_ref(), &token_out, amount_out);
95                 }
96                 // Even if send tokens fails, we don't return funds back to sender.
97                 PromiseOrValue::Value(U128(0))
98             }
99         }
100     }
101 }
```

Listing 2.13: src/token_receiver.rs

Suggestion I Check all the tokens listed in `actions` before the swapping to make sure no frozen tokens exist.

Feedback from the Project Will fix it in the next accumulated contract upgrade.

2.4 Notes

2.4.1 Delayed Price in Rated Swap Pool

Status

Introduced by `version 1`

Description Given the async nature of [NEAR](#) protocol, one transaction on the [NEAR](#) protocol may be executed in several blocks. The price of tokens in the [Rated Swap Pool](#) may not be the latest. Therefore, it should be noted that the token added to the [Rated Swap Pool](#) should be as stable as possible.

2.4.2 Timely Triggering `update_token_rate()`

Status

Introduced by [version 1](#)

Description Function `update_token_rate()` is used to get the newest rates of tokens from the token contracts and update them in the contract for further use. It's important for the team to make sure that the function will be triggered by the team timely.

2.4.3 Sensitive Functions Managed by DAO

Status Confirmed

Introduced by [version 1](#)

Description Privileged functions in [Ref-Exchange](#) are controlled by [DAO](#) (i.e., [ref-finance.sputnik-dao.near](#)). The [DAO](#) has the privilege to configure system parameters, change the state of the contract (pause and unpause), upgrade the contract, etc. The community should manage the [DAO](#) carefully.