Security Audit Report for Ref-Exchange

Date: Sept 28th, 2023

Version: 3.0

Contact: contact@blocksec.com

BlockSec
Audit Team

Digitally signed by
BlockSec Audit Team
Date: 2023.10.07
10:29:38 +08'00'

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

2.1.2 Lack of Storage Usage Check in function ft_on_transfer 6

2.2 DeFi Security 8

2.2.1 Unrestricted Referral Account 8

2.2.2 Incorrect Admin Fees Calculation in Simple Pool 9

2.3 Additional Recommendation 11

2.3.1 Lack of Check on Guardians' Removal 11

2.3.2 Two-Step Transfer of Privileged Account Ownership 12

2.3.3 Potential Elastic Supply Token Problem 12

2.3.4 Improper Check on the Admin Fees 12

2.3.5 Lack of Check in retrieve_unmanaged_token() 13

2.3.6 Lack of Check on the Gas Used by migrate() 14

2.3.7 Code Optimization (I) 15

2.3.8 Code Optimization (II) 18

2.3.9 Avoid Logging in View Functions 19

2.3.10 Slippage Protection in Function add_liquidity 20

2.4 Notes 23

2.4.1 Delayed Price in Rated Swap Pool 23

2.4.2 Timely Triggering update_token_rate() 23

2.4.3 Sensitive Functions Managed by DAO 23

Report Manifest

Item	Description
Client	Ref Finance
Target	Ref-Exchange

Version History

Version	Date	Description
1.0	November 2nd, 2022	First Version
2.0	November 20th, 2022	Second Version
3.0	September 28th, 2023	Third Version

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
	Version 2	19e98ec7e70b72d0a2bb1281eb8cd171cebcc931
	Version 3	edea28e1f9bb4f66f5f64eb8448f681f92ef3f10
	Version 4	422591c276224c6477cd638a88ab21807b4b5795

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:

```
ref-exchange
 |-- src
 |   |-- rated_swap
 |   |   |-- linear_rate.rs
 |   |   |-- math.rs
 |   |   |-- mod.rs
 |   |   |-- nearx_rate.rs
 |   |   |-- rate.rs
 |   |   |-- stnear.rs
 |   |
 |   |-- stable_swap
 |   |   |-- math.rs
 |   |   |-- mod.rs
 |   |
 |   |-- account_deposit.rs
 |   |-- action.rs
 |   |-- admin_fee.rs
 |   |-- custom_keys.rs
 |   |-- errors.rs
 |   |-- legacy.rs
 |   |-- lib.rs
 |   |-- multi_fungible_token.rs
```

[1] https://github.com/ref-finance/ref-contracts/tree/main/ref-exchange

```
|   |-- owner.rs
|   |-- pool.rs
|   |-- simple_pool.rs
|   |-- storage_impl.rs
|   |-- token_receiver.rs
|   |-- utils.rs
|   |-- 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.

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

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

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

		Likelihood	
		High	*Low*
Impact	*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.

Chapter 2 Findings

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

- High Risk: 0
- Medium Risk: 3
- Low Risk: 1
- Recommendations: 10
- Notes: 3

ID	Severity	Description	Category	Status
1	Medium	Improper Account Unregistration	Software Security	Fixed
2	Medium	Lack of Storage Usage Check in function ft_on_transfer	Software Security	Fixed
3	Low	Unrestricted Referral Account	DeFi Security	Fixed
4	Medium	Incorrect Admin Fees Calculation in Simple Pool	DeFi Security	Fixed
5	-	Lack of Check on Guardians' Removal	Recommendation	Fixed
6	-	Two-Step Transfer of Privileged Account Ownership	Recommendation	Confirmed
7	-	Potential Elastic Supply Token Problem	Recommendation	Confirmed
8	-	Improper Check on the Admin Fees	Recommendation	Fixed
9	-	Lack of Check in retrieve_unmanaged_token()	Recommendation	Confirmed
10	-	Lack of Check on the Gas Used by migrate()	Recommendation	Fixed
11	-	Code Optimization (I)	Recommendation	Fixed
12	-	Code Optimization (II)	Recommendation	Fixed
13	-	Avoid Logging in View Functions	Recommendation	Fixed
14	-	Slippage Protection in Function add_liquidity	Recommendation	Fixed*
15	-	Delayed Price in Rated Swap Pool	Note	Confirmed
16	-	Timely Triggering update_token_rate()	Note	Confirmed
17	-	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 Fixed in Version 2

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.

```
#[allow(unused_variables)]
#[payable]
fn storage_unregister(&mut self, force: Option<bool>) -> bool {
   assert_one_yocto();
   self.assert_contract_running();
   let account_id = env::predecessor_account_id();
   if let Some(account_deposit) = self.internal_get_account(&account_id) {
       // TODO: figure out force option logic.
       assert!(
           account_deposit.tokens.is_empty(),
           "{}", ERR18_TOKENS_NOT_EMPTY
       );
       self.accounts.remove(&account_id);
       Promise::new(account_id.clone()).transfer(account_deposit.near_amount);
       true
   } else {
       false
   }
}
```

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.

2.1.2 Lack of Storage Usage Check in function ft_on_transfer

Severity Medium

Status Fixed in `Version 4`

Introduced by `Version 3`

Description `HotZap` allows users to seamlessly swap tokens and provide liquidity to pools in a single transaction via the function `ft_on_transfer()`. However, adding liquidity may potentially increase the pool's storage usage. The function does not perform sufficient validation of user's storage usage.

```
    TokenReceiverMessage::HotZap {
        referral_id,
        hot_zap_actions,
        add_liquidity_infos
    } => {
        assert!(hot_zap_actions.len() > 0 && add_liquidity_infos.len() > 0);
        let sender_id: AccountId = sender_id.into();
        let mut account = self.internal_unwrap_account(&sender_id);
        let referral_id = referral_id.map(|x| x.to_string());
        let out_amounts = self.internal_direct_actions(
            token_in,
            amount.0,
            referral_id,
            &hot_zap_actions,
        );

        let mut token_cache = TokenCache::new();
        for (out_token_id, out_amount) in out_amounts {
```

```rust
            token_cache.add(&out_token_id, out_amount);
        }

        for add_liquidity_info in add_liquidity_infos {
            let mut pool = self.pools.get(add_liquidity_info.pool_id).expect(ERR85_NO_POOL);
            let tokens_in_pool = match &pool {
                Pool::SimplePool(p) => p.token_account_ids.clone(),
                Pool::RatedSwapPool(p) => p.token_account_ids.clone(),
                Pool::StableSwapPool(p) => p.token_account_ids.clone(),
            };

            let mut add_liquidity_amounts = add_liquidity_info.amounts.iter().map(|v| v.0).collect();

            match pool {
                Pool::SimplePool(_) => {
                    pool.add_liquidity(
                        &sender_id,
                        &mut add_liquidity_amounts,
                        false
                    );
                    if let Some(min_amounts) = add_liquidity_info.min_amounts {
                        // Check that all amounts are above request min amounts in case of front running that changes the exchange rate.
                        for (amount, min_amount) in add_liquidity_amounts.iter().zip(min_amounts.iter()) {
                            assert!(amount >= &min_amount.0, "{}", ERR86_MIN_AMOUNT);
                        }
                    }
                },
                Pool::StableSwapPool(_) | Pool::RatedSwapPool(_) => {
                    let min_shares = add_liquidity_info.min_shares.expect("Need input min_shares");
                    pool.add_stable_liquidity(
                        &sender_id,
                        &add_liquidity_amounts,
                        min_shares.into(),
                        AdminFees::new(self.admin_fee_bps),
                        false
                    );
                }
            };

            for (cost_token_id, cost_amount) in tokens_in_pool.iter().zip(add_liquidity_amounts.into_iter()) {
                token_cache.sub(cost_token_id, cost_amount);
            }

            self.pools.replace(add_liquidity_info.pool_id, &pool);
        }

        for (remain_token_id, remain_amount) in token_cache.0.iter() {
            account.deposit(remain_token_id, *remain_amount);
        }

        self.internal_save_account(&sender_id, account);

        env::log(
            format!(
                "HotZap remain internal account assets: {:?}",
                token_cache.0
            )
            .as_bytes(),
        );

        PromiseOrValue::Value(U128(0))
    }
```

Listing 2.2: src/token_receiver.rs

Impact The storage fees that should be claimed from users may be bypassed. Furthermore, users can claim extra fees with the function `remove_liquidity` or `remove_liquidity_by_tokens`.

Suggestion I Add storage check in the function `ft_on_transfer()`.

2.2 DeFi Security

2.2.1 Unrestricted Referral Account

Severity Low

Status Fixed in `Version 2`

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.

```
    #[allow(unreachable_code)]
    fn ft_on_transfer(
        &mut self,
        sender_id: ValidAccountId,
        amount: U128,
        msg: String,
    ) -> PromiseOrValue<U128> {
        self.assert_contract_running();
        let token_in = env::predecessor_account_id();
        // feature frozenlist
        self.assert_no_frozen_tokens(&[token_in.clone()]);
        if msg.is_empty() {
            // Simple deposit.
            self.internal_deposit(sender_id.as_ref(), &token_in, amount.into());
            PromiseOrValue::Value(U128(0))
        } else {
            // instant swap
            let message =
                serde_json::from_str::<TokenReceiverMessage>(&msg).expect(ERR28_WRONG_MSG_FORMAT);
            match message {
                TokenReceiverMessage::Execute {
                    referral_id,
                    actions,
                } => {
                    let referral_id = referral_id.map(|x| x.to_string());
                    let out_amounts = self.internal_direct_actions(
                        token_in,
                        amount.0,
                        referral_id,
                        &actions,
                    );
                    for (token_out, amount_out) in out_amounts.into_iter() {
                        self.internal_send_tokens(sender_id.as_ref(), &token_out, amount_out);
                    }
                    // Even if send tokens fails, we don't return funds back to sender.
                    PromiseOrValue::Value(U128(0))
                }
            }
        }
    }
```

Listing 2.3: 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 Fixed in `Version 2`

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:

$$ActualAdmin\_Fee = \frac{Total\_Share * Admin\_Fee * \frac{\sqrt{k'} - \sqrt{k}}{\sqrt{k'}}}{Total\_Share * Admin\_Fee * \frac{\sqrt{k'} - \sqrt{k}}{\sqrt{k'}} + Total\_Share} * \frac{\sqrt{k'}}{\sqrt{k'} - \sqrt{k}} = \frac{1}{\frac{1}{Admin\_Fee} + \frac{\sqrt{k'} - \sqrt{k}}{\sqrt{k'}}}$$

, 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:

$$Minted\_Share = Total\_Share * \frac{\sqrt{k'} - \sqrt{k}}{(\frac{1}{Admin\_Fee} - 1) * \sqrt{k'} + \sqrt{k}}$$

```rust
pub fn swap(
    &mut self,
    token_in: &AccountId,
    amount_in: Balance,
    token_out: &AccountId,
    min_amount_out: Balance,
    admin_fee: &AdminFees,
) -> Balance {
    assert_ne!(token_in, token_out, "{}", ERR73_SAME_TOKEN);
    let in_idx = self.token_index(token_in);
    let out_idx = self.token_index(token_out);
    let amount_out = self.internal_get_return(in_idx, amount_in, out_idx);
    assert!(amount_out >= min_amount_out, "{}", ERR68_SLIPPAGE);
    env::log(
        format!(
            "Swapped {} {} for {} {}",
            amount_in, token_in, amount_out, token_out
        )
        .as_bytes(),
    );

    let prev_invariant =
        integer_sqrt(U256::from(self.amounts[in_idx]) * U256::from(self.amounts[out_idx]));

    self.amounts[in_idx] += amount_in;
    self.amounts[out_idx] -= amount_out;

    // "Invariant" is by how much the dot product of amounts increased due to fees.
    let new_invariant =
        integer_sqrt(U256::from(self.amounts[in_idx]) * U256::from(self.amounts[out_idx]));

    // Invariant can not reduce (otherwise loosing balance of the pool and something it broken).
    assert!(new_invariant >= prev_invariant, "{}", ERR75_INVARIANT_REDUCE);
    let numerator = (new_invariant - prev_invariant) * U256::from(self.shares_total_supply);

    // Allocate exchange fee as fraction of total fee by issuing LP shares proportionally.
    if admin_fee.exchange_fee > 0 && numerator > U256::zero() {
        let denominator = new_invariant * FEE_DIVISOR / admin_fee.exchange_fee;
        self.mint_shares(&admin_fee.exchange_id, (numerator / denominator).as_u128());
    }

    // If there is referral provided and the account already registered LP, allocate it % of LP rewards.
```

```
        if let Some(referral_id) = &admin_fee.referral_id {
            if admin_fee.referral_fee > 0
                && numerator > U256::zero()
                && self.shares.contains_key(referral_id)
            {
                let denominator = new_invariant * FEE_DIVISOR / admin_fee.referral_fee;
                self.mint_shares(referral_id, (numerator / denominator).as_u128());
            }
        }

        // Keeping track of volume per each input traded separately.
        // Reported volume with fees will be sum of input, without fees will be sum of output.
        self.volumes[in_idx].input.0 += amount_in;
        self.volumes[in_idx].output.0 += amount_out;

        amount_out
    }
```

Listing 2.4: 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.

Note After the patch, the fee mechanism among `Simple Pool`, `Stable Pool`, and `Rated Pool` are consistent. However, the actual admin fee rate is higher than the `admin_fee_bps` configured in `AdminFees`. The reason is that the `LP` fees are distributed to all shares in the pool including the newly minted shares for admins.

2.3 Additional Recommendation

2.3.1 Lack of Check on Guardians' Removal

Status Fixed in `Version 2`

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.

```
    #[payable]
    pub fn remove_guardians(&mut self, guardians: Vec<ValidAccountId>) {
        assert_one_yocto();
        self.assert_owner();
        for guardian in guardians {
            self.guardians.remove(guardian.as_ref());
        }
    }
```

Listing 2.5: 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.

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

Listing 2.6: 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 Fixed in Version 2

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.

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

Listing 2.7: 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.

```
    #[payable]
    pub fn retrieve_unmanaged_token(&mut self, token_id: ValidAccountId, amount: U128) -> Promise {
        self.assert_owner();
        assert_one_yocto();
        let token_id: AccountId = token_id.into();
        let amount: u128 = amount.into();
        assert!(amount > 0, "{}", ERR29_ILLEGAL_WITHDRAW_AMOUNT);
        env::log(
            format!(
                "Going to retrieve token {} to owner, amount: {}",
                &token_id, amount
            )
            .as_bytes(),
        );
        ext_fungible_token::ft_transfer(
            self.owner_id.clone(),
            U128(amount),
            None,
            &token_id,
            1,
            env::prepaid_gas() - GAS_FOR_BASIC_OP,
        )
    }
```

Listing 2.8: 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 Fixed in `Version 2`

Introduced by `Version 1`

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

```
#[no_mangle]
pub extern "C" fn upgrade() {
    env::setup_panic_hook();
    env::set_blockchain_interface(Box::new(near_blockchain::NearBlockchain {}));
    let contract: Contract = env::state_read().expect(ERR103_NOT_INITIALIZED);
    contract.assert_owner();
    let current_id = env::current_account_id().into_bytes();
    let method_name = "migrate".as_bytes().to_vec();
    unsafe {
        BLOCKCHAIN_INTERFACE.with(|b| {
            // Load input into register 0.
            b.borrow()
                .as_ref()
                .expect(BLOCKCHAIN_INTERFACE_NOT_SET_ERR)
                .input(0);
            let promise_id = b
                .borrow()
                .as_ref()
                .expect(BLOCKCHAIN_INTERFACE_NOT_SET_ERR)
                .promise_batch_create(current_id.len() as _, current_id.as_ptr() as _);
            b.borrow()
                .as_ref()
                .expect(BLOCKCHAIN_INTERFACE_NOT_SET_ERR)
                .promise_batch_action_deploy_contract(promise_id, u64::MAX as _, 0);
            let attached_gas = env::prepaid_gas() - env::used_gas() - GAS_FOR_MIGRATE_CALL;
            b.borrow()
                .as_ref()
                .expect(BLOCKCHAIN_INTERFACE_NOT_SET_ERR)
                .promise_batch_action_function_call(
                    promise_id,
                    method_name.len() as _,
                    method_name.as_ptr() as _,
                    0 as _,
                    0 as _,
                    0 as _,
                    attached_gas,
                );
        });
    }
}
```

Listing 2.9: 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 Fixed in `Version 2`

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
                   changes the exchange rate.
262            for (amount, min_amount) in amounts.iter().zip(min_amounts.iter()) {
263                assert!(amount >= &min_amount.0, "{}", ERR86_MIN_AMOUNT);
```

```
            }
        }
        let mut deposits = self.internal_unwrap_or_default_account(&sender_id);
        let tokens = pool.tokens();
        // Subtract updated amounts from deposits. This will fail if there is not enough funds for any of the tokens.
        for i in 0..tokens.len() {
            deposits.withdraw(&tokens[i], amounts[i]);
        }
        self.internal_save_account(&sender_id, deposits);
        self.pools.replace(pool_id, &pool);
        self.internal_check_storage(prev_storage);

        U128(shares)
    }
```

Listing 2.10: src/lib.rs

```
    #[payable]
    pub fn add_stable_liquidity(
        &mut self,
        pool_id: u64,
        amounts: Vec<U128>,
        min_shares: U128,
    ) -> U128 {
        self.assert_contract_running();
        assert!(
            env::attached_deposit() > 0,
            "{}", ERR35_AT_LEAST_ONE_YOCTO
        );
        let prev_storage = env::storage_usage();
        let sender_id = env::predecessor_account_id();
        let amounts: Vec<u128> = amounts.into_iter().map(|amount| amount.into()).collect();
        let mut pool = self.pools.get(pool_id).expect(ERR85_NO_POOL);
        // feature frozenlist
        self.assert_no_frozen_tokens(pool.tokens());
        // Add amounts given to liquidity first. It will return the balanced amounts.
        let mint_shares = pool.add_stable_liquidity(
            &sender_id,
            &amounts,
            min_shares.into(),
            AdminFees::new(self.exchange_fee),
        );
        let mut deposits = self.internal_unwrap_or_default_account(&sender_id);
        let tokens = pool.tokens();
        // Subtract amounts from deposits. This will fail if there is not enough funds for any of the tokens.
        for i in 0..tokens.len() {
            deposits.withdraw(&tokens[i], amounts[i]);
        }
        self.internal_save_account(&sender_id, deposits);
        self.pools.replace(pool_id, &pool);
        self.internal_check_storage(prev_storage);

        mint_shares.into()
    }
```

Listing 2.11: src/lib.rs

```
    #[payable]
    pub fn remove_liquidity(&mut self, pool_id: u64, shares: U128, min_amounts: Vec<U128>) -> Vec<
        U128> {
        assert_one_yocto();
        self.assert_contract_running();
        let prev_storage = env::storage_usage();
        let sender_id = env::predecessor_account_id();
        let mut pool = self.pools.get(pool_id).expect(ERR85_NO_POOL);
        // feature frozenlist
        self.assert_no_frozen_tokens(pool.tokens());
        let amounts = pool.remove_liquidity(
            &sender_id,
            shares.into(),
            min_amounts
                .into_iter()
                .map(|amount| amount.into())
                .collect(),
        );
        self.pools.replace(pool_id, &pool);
        let tokens = pool.tokens();
        let mut deposits = self.internal_unwrap_or_default_account(&sender_id);
        for i in 0..tokens.len() {
            deposits.deposit(&tokens[i], amounts[i]);
        }
        // Freed up storage balance from LP tokens will be returned to near_balance.
        if prev_storage > env::storage_usage() {
            deposits.near_amount +=
                (prev_storage - env::storage_usage()) as Balance * env::storage_byte_cost();
        }
        self.internal_save_account(&sender_id, deposits);

        amounts
            .into_iter()
            .map(|amount| amount.into())
            .collect()
    }
```

Listing 2.12: src/lib.rs

```
    #[payable]
    pub fn remove_liquidity_by_tokens(
        &mut self, pool_id: u64,
        amounts: Vec<U128>,
        max_burn_shares: U128
    ) -> U128 {
        assert_one_yocto();
        self.assert_contract_running();
        let prev_storage = env::storage_usage();
        let sender_id = env::predecessor_account_id();
        let mut pool = self.pools.get(pool_id).expect(ERR85_NO_POOL);
        // feature frozenlist
        self.assert_no_frozen_tokens(pool.tokens());
        let burn_shares = pool.remove_liquidity_by_tokens(
            &sender_id,
            amounts
                .clone()
                .into_iter()
                .map(|amount| amount.into())
                .collect(),
            max_burn_shares.into(),
            AdminFees::new(self.exchange_fee),
        );
        self.pools.replace(pool_id, &pool);
        let tokens = pool.tokens();
        let mut deposits = self.internal_unwrap_or_default_account(&sender_id);
        for i in 0..tokens.len() {
            deposits.deposit(&tokens[i], amounts[i].into());
        }
        // Freed up storage balance from LP tokens will be returned to near_balance.
        if prev_storage > env::storage_usage() {
            deposits.near_amount +=
                (prev_storage - env::storage_usage()) as Balance * env::storage_byte_cost();
        }
        self.internal_save_account(&sender_id, deposits);

        burn_shares.into()
    }
```

Listing 2.13: 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 Fixed in `Version 2`

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.

```
    #[allow(unreachable_code)]
    fn ft_on_transfer(
        &mut self,
        sender_id: ValidAccountId,
        amount: U128,
        msg: String,
    ) -> PromiseOrValue<U128> {
        self.assert_contract_running();
        let token_in = env::predecessor_account_id();
        // feature frozenlist
        self.assert_no_frozen_tokens(&[token_in.clone()]);
        if msg.is_empty() {
            // Simple deposit.
            self.internal_deposit(sender_id.as_ref(), &token_in, amount.into());
            PromiseOrValue::Value(U128(0))
        } else {
            // instant swap
            let message =
                serde_json::from_str::<TokenReceiverMessage>(&msg).expect(ERR28_WRONG_MSG_FORMAT);
            match message {
                TokenReceiverMessage::Execute {
                    referral_id,
                    actions,
                } => {
                    let referral_id = referral_id.map(|x| x.to_string());
                    let out_amounts = self.internal_direct_actions(
                        token_in,
                        amount.0,
                        referral_id,
                        &actions,
                    );
                    for (token_out, amount_out) in out_amounts.into_iter() {
                        self.internal_send_tokens(sender_id.as_ref(), &token_out, amount_out);
                    }
                    // Even if send tokens fails, we don't return funds back to sender.
                    PromiseOrValue::Value(U128(0))
                }
            }
        }
    }
```

Listing 2.14: 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.

2.3.9 Avoid Logging in View Functions

Status Fixed in `Version 4`

Introduced by `Version 3`

Description The function shown below will always emit the logs regardless of whether the `is_view` argument is `true` or `false`, which can lead to inaccuracies in off-chain statistics and analytics. Though logs emitted in most of the functions can be differentiated by originating from a view account or not, logs emitted in function swap cannot be differentiated between view and non-view usage.

File	Function
simple_pool.rs	add_liquidity
simple_pool.rs	remove_liquidity
simple_pool.rs	swap
rated_swap/mod.rs	add_liquidity
rated_swap/mod.rs	remove_liquidity_by_shares
rated_swap/mod.rs	remove_liquidity_by_tokens
rated_swap/mod.rs	swap
stable_swap/mod.rs	add_liquidity
stable_swap/mod.rs	remove_liquidity_by_shares
stable_swap/mod.rs	remove_liquidity_by_tokens
stable_swap/mod.rs	swap

Suggestion I Avoid emitting logs when is_view is true.

2.3.10 Slippage Protection in Function add_liquidity

Status Fixed* in Version 4

Introduced by Version 3

Description When providing liquidity to simple pools via the operation HotZap, users can optionally specify the parameter min_amounts to control slippage. However, when providing liquidity to stable pools or rated pools, the function add_liqudity() requires users to provide a mint_shares.

Considering the consistency and risk of frontrunning, it's recommended to check that min_amounts has to be provided.

```rust
    #[payable]
    pub fn add_liquidity(
        &mut self,
        pool_id: u64,
        amounts: Vec<U128>,
        min_amounts: Option<Vec<U128>>,
    ) -> U128 {
        self.assert_contract_running();
        assert!(env::attached_deposit() > 0, "{}", ERR35_AT_LEAST_ONE_YOCTO);
        let prev_storage = env::storage_usage();
        let sender_id = env::predecessor_account_id();
        let mut amounts: Vec<u128> = amounts.into_iter().map(|amount| amount.into()).collect();
        let mut pool = self.pools.get(pool_id).expect(ERR85_NO_POOL);
        // feature frozenlist
        self.assert_no_frozen_tokens(pool.tokens());
        // Add amounts given to liquidity first. It will return the balanced amounts.
        let shares = pool.add_liquidity(&sender_id, &mut amounts, false);
        if let Some(min_amounts) = min_amounts {
            // Check that all amounts are above request min amounts in case of front running that changes the exchange rate.
            for (amount, min_amount) in amounts.iter().zip(min_amounts.iter()) {
                assert!(amount >= &min_amount.0, "{}", ERR86_MIN_AMOUNT);
            }
        }
        // [AUDITION_AMENDMENT] 2.3.7 Code Optimization (I)
        let mut deposits = self.internal_unwrap_account(&sender_id);
        let tokens = pool.tokens();
        // Subtract updated amounts from deposits. This will fail if there is not enough funds for any of the tokens.
        for i in 0..tokens.len() {
            deposits.withdraw(&tokens[i], amounts[i]);
        }
        self.internal_save_account(&sender_id, deposits);
        self.pools.replace(pool_id, &pool);
        self.internal_check_storage(prev_storage);


        U128(shares)
    }
```

Listing 2.15: src/lib.rs

```
TokenReceiverMessage::HotZap {
    referral_id,
    hot_zap_actions,
    add_liquidity_infos
} => {
    assert!(hot_zap_actions.len() > 0 && add_liquidity_infos.len() > 0);
    let sender_id: AccountId = sender_id.into();
    let mut account = self.internal_unwrap_account(&sender_id);
    let referral_id = referral_id.map(|x| x.to_string());
    let out_amounts = self.internal_direct_actions(
        token_in,
        amount.0,
        referral_id,
        &hot_zap_actions,
    );

    let mut token_cache = TokenCache::new();
    for (out_token_id, out_amount) in out_amounts {
        token_cache.add(&out_token_id, out_amount);
    }

    for add_liquidity_info in add_liquidity_infos {
        let mut pool = self.pools.get(add_liquidity_info.pool_id).expect(ERR85_NO_POOL);
        let tokens_in_pool = match &pool {
            Pool::SimplePool(p) => p.token_account_ids.clone(),
            Pool::RatedSwapPool(p) => p.token_account_ids.clone(),
            Pool::StableSwapPool(p) => p.token_account_ids.clone(),
        };

        let mut add_liquidity_amounts = add_liquidity_info.amounts.iter().map(|v| v.0).collect();

        match pool {
            Pool::SimplePool(_) => {
                pool.add_liquidity(
                    &sender_id,
                    &mut add_liquidity_amounts,
                    false
                );
                if let Some(min_amounts) = add_liquidity_info.min_amounts {
                    // Check that all amounts are above request min amounts in case of front running that changes the exchange rate.
                    for (amount, min_amount) in add_liquidity_amounts.iter().zip(min_amounts.iter()) {
                        assert!(amount >= &min_amount.0, "{}", ERR86_MIN_AMOUNT);
                    }
                }
            },
            Pool::StableSwapPool(_) | Pool::RatedSwapPool(_) => {
                let min_shares = add_liquidity_info.min_shares.expect("Need input min_shares");
                pool.add_stable_liquidity(
                    &sender_id,
                    &add_liquidity_amounts,
                    min_shares.into(),
                    AdminFees::new(self.admin_fee_bps),
                    false
                );
            }
        };

        for (cost_token_id, cost_amount) in tokens_in_pool.iter().zip(add_liquidity_amounts.into_iter()) {
            token_cache.sub(cost_token_id, cost_amount);
        }

        self.pools.replace(add_liquidity_info.pool_id, &pool);
    }

    for (remain_token_id, remain_amount) in token_cache.0.iter() {
        account.deposit(remain_token_id, *remain_amount);
    }

    self.internal_save_account(&sender_id, account);

    env::log(
        format!(
            "HotZap remain internal account assets: {:?}",
            token_cache.0
        )
        .as_bytes(),
    );

    PromiseOrValue::Value(U128(0))
}
```

Listing 2.16: src/token_receiver.rs

Suggestion I Revise the logic accordingly.

Feedback from the Project The add_liquidity function has not been updated for compatibility.

2.4 Notes

2.4.1 Delayed Price in Rated Swap Pool

Status Confirmed

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 Confirmed

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.