BLOCKSEC

Security Audit
Report for Near Staker

Date: September 04, 2024 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	TruFin
Target	Near Staker

Version History

Version	Date	Description
1.0	September 04, 2024	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-09-04 09:51:42+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 Near Staker[1] of TruFin.

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
Near Staker	`Version 1`	`ee21a4e7a3c7c8208385ff83ab87c86188e4239b`
	`Version 2`	`624ab945bc9223d3f24dd78291a6a3e0bbab2174`

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/TruFin-io/near-staker-audit

- **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 **seven** potential issues. Besides, we also have **three** recommendations and **three** notes as follows:

- High Risk: 2
- Medium Risk: 2
- Low Risk: 3
- Recommendation: 3
- Note: 3

ID	Severity	Description	Category	Status
1	High	Potential DoS in function `send_unstake_promises()`	DeFi Security	Fixed
2	High	Potential incorrect update on `pool.last_unstake`	DeFi Security	Fixed
3	Low	Incorrect gas usage in function `send_stake_promises()`	DeFi Security	Fixed
4	Medium	Incorrect share price calculation due to asynchronous updates of `total_staked` and `total_supply`	DeFi Security	Fixed
5	Medium	Lack of callback function `ft_resolve_transfer()`	DeFi Security	Fixed
6	Low	Lack of timely account registration for treasury in function `set_treasury()`	DeFi Security	Fixed
7	Low	Potential unrefunded fee in function `distribute_all()`	DeFi Security	Fixed
8	-	Lack of check in function `check_owner()`	Recommendation	Confirmed
9	-	Redundant code	Recommendation	Fixed
10	-	Using constants in function `internal_deposit_and_stake()`	Recommendation	Fixed
11	-	Potential centralization risk	Note	
12	-	Potential discrepancies in share price calculations	Note	
13	-	Fixed token of distribution fee	Note	

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Potential DoS in function `send_unstake_promises()`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The contract will lock itself when the user performs an unstake operation (in function `unstake()`). However, when the `shares_amount` to be burned is 0, the function `send_unstake_promises()` directly refunds the attached `NEAR` without unlocking the contract, which leads to a potential DoS of the contract.

```
618   /// Unstakes NEAR from default pool.
619   #[payable]
620   pub fn unstake(&mut self, amount: U128) -> Promise {
621       self.check_not_paused();
622       self.check_not_locked();
623       self.is_locked = true;
624
625
626       self.check_whitelisted();
627
628
629       self.internal_unstake(
630           self.default_delegation_pool.clone(),
631           amount.0,
632           env::predecessor_account_id(),
633       )
634   }
```

Listing 2.1: lib.rs

```
204   /// Unstakes the specified amount of NEAR tokens from the specified delegation pool.
205   pub(crate) fn internal_unstake(
206       &mut self,
207       pool_id: AccountId,
208       amount: u128,
209       caller: AccountId,
210   ) -> Promise {
211       self.check_contract_in_sync();
212
213
214       let attached_near = env::attached_deposit();
215       require!(
216           attached_near.as_yoctonear() >= Self::get_storage_cost().0,
217           ERR_STORAGE_DEPOSIT_TOO_SMALL
218       );
219
```

```
220
221     // We must check that there is no pending unstake from previous epochs on the pool. If
            there is, we cannot unlock as
222     // it would push back the pending unstake by a further four epochs.
223     let pool_last_unstake = self.delegation_pools.get(&pool_id).unwrap().last_unstake;
224     let current_epoch = env::epoch_height();
225
226
227     // we can unlock if the last unstake happened in the same epoch or more than 4 epochs ago (
            there is withdrawable stake)
228     if let Some(last_unstake) = pool_last_unstake {
229         require!(
230             last_unstake == current_epoch
231                 || last_unstake + NUM_EPOCHS_TO_UNLOCK <= current_epoch,
232             ERR_UNSTAKE_LOCKED
233         );
234     }
235
236
237     // if the total staked is up to date, check the requested unstake amount
238     let amount = self.internal_check_unstake_amount(&pool_id, amount, &caller);
239
240
241     self.send_unstake_promises(pool_id, amount, caller, attached_near)
242 }
```

Listing 2.2: internal.rs

```
118 /// Unstakes NEAR from the specified pool, withdrawing first if necessary.
119 pub(crate) fn send_unstake_promises(
120     &mut self,
121     pool_id: AccountId,
122     amount: u128,
123     caller: AccountId,
124     attached_near: NearToken,
125 ) -> Promise {
126     // ensure amount of shares burned is greater than 0
127     let (share_price_num, share_price_denom) = Self::internal_share_price(
128         self.total_staked,
129         self.token.ft_total_supply().0,
130         self.tax_exempt_stake,
131         self.fee,
132     );
133
134
135     let shares_amount =
136         Self::convert_to_shares(amount, share_price_num, share_price_denom, false);
137     if shares_amount == 0 {
138         log!("Failed to unstake: {}", ERR_UNSTAKE_AMOUNT_TOO_LOW);
139         return Promise::new(caller).transfer(attached_near);
140     }
141
142
```

```
143         // burn user shares
144         self.internal_burn(shares_amount, caller.clone());
145
146
147         // prepare unstake arguments
148         let unstake_amount = json!({ "amount": NearToken::from_yoctonear(amount) })
149             .to_string()
150             .into_bytes();
151
152
153         let staker_id_arg = json!({ "account_id": env::current_account_id()})
154             .to_string()
155             .into_bytes();
156
157
158         let pre_unstake_staker_balance = env::account_balance();
159         let mut promise = Promise::new(pool_id.clone());
160
161
162         // we fetch the total amount requested for unstake on the given pool and last unstake epoch
                as we should withdraw
163         // any unlocked stake into the staker before unlocking more due to the 4 epoch wait period
164         let pool_info = self.delegation_pools.get(&pool_id).unwrap();
165         let mut withdraw_occurred: bool = false;
166
167
168         if let Some(last_unstake) = pool_info.last_unstake {
169             if last_unstake + NUM_EPOCHS_TO_UNLOCK <= env::epoch_height()
170                 && pool_info.total_unstaked.0 > 0
171             {
172                 // if there is stake to withdraw, we withdraw it before calling unstake
173                 let withdraw_args = json!({ "amount": pool_info.total_unstaked })
174                     .to_string()
175                     .into_bytes();
176                 promise = promise.function_call(
177                     "withdraw".to_owned(),
178                     withdraw_args,
179                     NO_DEPOSIT,
180                     XCC_GAS,
181                 );
182                 withdraw_occurred = true;
183             }
184         }
185         // call unstake on the pool and fetch the new account unstaked balance
186         promise = promise
187             .function_call("unstake".to_owned(), unstake_amount, NO_DEPOSIT, XCC_GAS)
188             .function_call(
189                 "get_account_unstaked_balance".to_owned(),
190                 staker_id_arg,
191                 NO_DEPOSIT,
192                 VIEW_GAS,
193             );
194         promise.then(
```

```
195             Self::ext(env::current_account_id())
196                 .with_static_gas(XCC_GAS)
197                 .finalize_unstake(
198                     pool_id,
199                     U128(amount),
200                     caller,
201                     pre_unstake_staker_balance,
202                     share_price_num.to_string(),
203                     share_price_denom.to_string(),
204                     U128(shares_amount),
205                     withdraw_occurred,
206                     attached_near,
207                 ),
208         )
209     }
```

Listing 2.3: internal.rs

Impact Unless the privileged admin manually invokes the function `manual_unlock()` to unlock the contract, most of the functions will be locked, leading to DoS.

Suggestion Unlock the contract in the specified condition.

2.1.2 Potential incorrect update on `pool.last_unstake`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `finalize_unstake()`, `pool.last_unstake` is updated to the `epoch_height` of the current epoch. However, since the execution of the function `unstake()` spans multiple blocks, if the `unlock()` function is invoked at the end of the previous epoch, function `finalize_unstake()` can be executed at the beginning of the new epoch, causing `pool.last_unstake` to be updated to an incorrect `epoch_height`.

```
1097    #[private]
1098    /// Handles the unstake promise, performing associated accounting if successful.
1099    pub fn finalize_unstake(
1100        &mut self,
1101        pool_id: AccountId,
1102        amount: U128,
1103        caller: AccountId,
1104        pre_unstake_staker_balance: NearToken,
1105        share_price_num: String,
1106        share_price_denom: String,
1107        shares_amount: U128,
1108        withdraw_occurred: bool,
1109        attached_near: NearToken,
1110        #[callback_result] new_unstaked_amount: Result<U128, PromiseError>,
1111    ) {
1112        self.is_locked = false;
1113
```

```
1114
1115        let new_unstaked_amount = match new_unstaked_amount {
1116            Ok(amount) => amount.0,
1117            Err(_) => {
1118                log!("Failed to unstake: {}", ERR_CALLBACK_FAILED);
1119                self.internal_mint(shares_amount.0, caller.clone());
1120                Promise::new(caller).transfer(attached_near);
1121                return;
1122            }
1123        };
1124        let pool = self.delegation_pools.get_mut(&pool_id).unwrap();
1125
1126
1127        if withdraw_occurred {
1128            self.withdrawn_amount += pool.total_unstaked.0;
1129            // if a withdraw occurred, the new total unstake amount on the pool should be the
                amount
1130            // requested in this unstake.
1131            pool.total_unstaked = amount;
1132        } else {
1133            // if no withdraw occurred we add the requested unstake amount to the pool total
                unstaked amount
1134            pool.total_unstaked = (pool.total_unstaked.0 + amount.0).into();
1135        }
1136
1137
1138        // update delegation pool and total_staked
1139        pool.last_unstake = Some(env::epoch_height());
1140        pool.total_staked = (pool.total_staked.0 - amount.0).into();
1141        self.total_staked -= amount.0;
1142        self.tax_exempt_stake = self.tax_exempt_stake.saturating_sub(amount.0);
1143        log!("Updated total_staked: {}", self.total_staked);
1144
1145
1146        log!(
1147            "New unstaked amount {}. Pool total unstaked {}. Was withdrawn: {}. Pre balance {}.
                Post balance {}",
1148            new_unstaked_amount,
1149            pool.total_unstaked.0,
1150            withdraw_occurred,
1151            pre_unstake_staker_balance,
1152            env::account_balance()
1153        );
1154
1155
1156        // create the unstake request
1157        self.unstake_nonce += 1;
1158
1159
1160        let unstake_request = UnstakeRequest {
1161            pool_id: pool_id.clone(),
1162            near_amount: amount.0,
1163            user: caller.clone(),
```

```
1164            epoch: env::epoch_height(),
1165        };
1166
1167
1168        self.unstake_requests
1169            .insert(self.unstake_nonce, unstake_request);
1170
1171
1172        // refund any excess NEAR to allocator
1173        let storage_cost = NearToken::from_yoctonear(Self::get_storage_cost().0);
1174        if attached_near > storage_cost {
1175            Promise::new(caller.clone()).transfer(attached_near.checked_sub(storage_cost).unwrap())
                    ;
1176        }
1177
1178
1179        // emit Unstaked event
1180        Event::UnstakedEvent {
1181            user_id: &caller,
1182            amount: &amount,
1183            user_balance: &U128(self.token.accounts.get(&caller).unwrap_or(0)),
1184            shares_amount: &shares_amount,
1185            total_staked: &U128(self.total_staked),
1186            total_supply: &U128(self.token.total_supply),
1187            share_price_num: &share_price_num,
1188            share_price_denom: &share_price_denom,
1189            unstake_nonce: &U128(self.unstake_nonce),
1190            epoch: &env::epoch_height().into(),
1191            pool_id: &pool_id,
1192        }
1193        .emit();
1194    }
```

Listing 2.4: lib.rs

Impact Users can invoke function `unstake()` every epoch.

Suggestion Update `pool.last_unstake` before the cross contract invocation.

2.1.3 Incorrect gas usage in function `send_stake_promises()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `send_stake_promises()`, two cross-contract calls are constructed sequentially. The first one invokes the function `deposit_and_stake()` of the `staking_pool` to deposit the `NEAR` submitted by the user into the `staking_pool`. The second one invokes `get_account_total_ba` to query the balance after the deposit. However, the gas attached to the second cross-contract call is incorrect. Specifically, since `get_account_total_balance()` is a view function, it should use `VIEW_GAS` instead of `XCC_GAS`.

```rust
pub(crate) fn send_stake_promises(
    pool_id: AccountId,
    amount: u128,
    caller: AccountId,
) -> Promise {
    let staker_id: AccountId = env::current_account_id();


    let staker_arg = json!({ "account_id": staker_id }).to_string().into_bytes();


    // we first call deposit_and_stake followed by get_account_total_balance to ensure the
        stake has been added
    Promise::new(pool_id.clone())
        .function_call(
            "deposit_and_stake".to_owned(),
            NO_ARGS,
            NearToken::from_yoctonear(amount),
            Gas::from_tgas(30),
        )
        .function_call(
            "get_account_total_balance".to_owned(),
            staker_arg,
            NO_DEPOSIT,
            XCC_GAS,
        )
        .then(
            Self::ext(env::current_account_id())
                .with_static_gas(XCC_GAS)
                .finalize_deposit_and_stake(pool_id, U128(amount), caller),
        )
}
```

Listing 2.5: internal.rs

```rust
pub const XCC_GAS: Gas = Gas::from_tgas(30); // approx gas needed for cross-contract calls
pub const VIEW_GAS: Gas = Gas::from_tgas(5); // approx gas needed for view calls
```

Listing 2.6: constants.rs

Impact The invoker overpaid for gas.

Suggestion Replace `XCC_GAS` with `VIEW_GAS`.

2.1.4 Incorrect share price calculation due to asynchronous updates of `total_staked` and `total_supply`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description

During the unstaking process, the user's shares would first be burned, and then `total_staked` will be decreased in the callback function. This would cause the share price to temporarily rise before the callback is executed. However, operations such as `allocate()` and `distribute_reward()` can be executed without the contract being in an `unlocked` state, and they all depend on the share price for calculation. Therefore, if these operations are executed when the share price becomes inflated before the callback, it would yield inaccurate results. In addition, these operations (e.g., `allocate()`) do not invoke the function `check_contract_in_sync()` to validate that the contract state (i.e., `total_staked`) is in sync before executing, which would also affect the accuracy of the operations.

```
118   /// Unstakes NEAR from the specified pool, withdrawing first if necessary.
119   pub(crate) fn send_unstake_promises(
120       &mut self,
121       pool_id: AccountId,
122       amount: u128,
123       caller: AccountId,
124       attached_near: NearToken,
125   ) -> Promise {
126       // ensure amount of shares burned is greater than 0
127       let (share_price_num, share_price_denom) = Self::internal_share_price(
128           self.total_staked,
129           self.token.ft_total_supply().0,
130           self.tax_exempt_stake,
131           self.fee,
132       );
133
134
135       let shares_amount =
136           Self::convert_to_shares(amount, share_price_num, share_price_denom, false);
137       if shares_amount == 0 {
138           log!("Failed to unstake: {}", ERR_UNSTAKE_AMOUNT_TOO_LOW);
139           return Promise::new(caller).transfer(attached_near);
140       }
141
142
143       // burn user shares
144       self.internal_burn(shares_amount, caller.clone());
145
146
147       // prepare unstake arguments
148       let unstake_amount = json!({ "amount": NearToken::from_yoctonear(amount) })
149           .to_string()
150           .into_bytes();
151
152
153       let staker_id_arg = json!({ "account_id": env::current_account_id()})
154           .to_string()
155           .into_bytes();
156
157
158       let pre_unstake_staker_balance = env::account_balance();
159       let mut promise = Promise::new(pool_id.clone());
```

```
160
161
162         // we fetch the total amount requested for unstake on the given pool and last unstake epoch
                as we should withdraw
163         // any unlocked stake into the staker before unlocking more due to the 4 epoch wait period
164         let pool_info = self.delegation_pools.get(&pool_id).unwrap();
165         let mut withdraw_occurred: bool = false;
166
167
168         if let Some(last_unstake) = pool_info.last_unstake {
169             if last_unstake + NUM_EPOCHS_TO_UNLOCK <= env::epoch_height()
170                 && pool_info.total_unstaked.0 > 0
171             {
172                 // if there is stake to withdraw, we withdraw it before calling unstake
173                 let withdraw_args = json!({ "amount": pool_info.total_unstaked })
174                     .to_string()
175                     .into_bytes();
176                 promise = promise.function_call(
177                     "withdraw".to_owned(),
178                     withdraw_args,
179                     NO_DEPOSIT,
180                     XCC_GAS,
181                 );
182                 withdraw_occurred = true;
183             }
184         }
185         // call unstake on the pool and fetch the new account unstaked balance
186         promise = promise
187             .function_call("unstake".to_owned(), unstake_amount, NO_DEPOSIT, XCC_GAS)
188             .function_call(
189                 "get_account_unstaked_balance".to_owned(),
190                 staker_id_arg,
191                 NO_DEPOSIT,
192                 VIEW_GAS,
193             );
194         promise.then(
195             Self::ext(env::current_account_id())
196                 .with_static_gas(XCC_GAS)
197                 .finalize_unstake(
198                     pool_id,
199                     U128(amount),
200                     caller,
201                     pre_unstake_staker_balance,
202                     share_price_num.to_string(),
203                     share_price_denom.to_string(),
204                     U128(shares_amount),
205                     withdraw_occurred,
206                     attached_near,
207                 ),
208         )
209     }
```

Listing 2.7: internal.rs

```
1097    #[private]
1098    /// Handles the unstake promise, performing associated accounting if successful.
1099    pub fn finalize_unstake(
1100        &mut self,
1101        pool_id: AccountId,
1102        amount: U128,
1103        caller: AccountId,
1104        pre_unstake_staker_balance: NearToken,
1105        share_price_num: String,
1106        share_price_denom: String,
1107        shares_amount: U128,
1108        withdraw_occurred: bool,
1109        attached_near: NearToken,
1110        #[callback_result] new_unstaked_amount: Result<U128, PromiseError>,
1111    ) {
1112        self.is_locked = false;
1113
1114
1115        let new_unstaked_amount = match new_unstaked_amount {
1116            Ok(amount) => amount.0,
1117            Err(_) => {
1118                log!("Failed to unstake: {}", ERR_CALLBACK_FAILED);
1119                self.internal_mint(shares_amount.0, caller.clone());
1120                Promise::new(caller).transfer(attached_near);
1121                return;
1122            }
1123        };
1124        let pool = self.delegation_pools.get_mut(&pool_id).unwrap();
1125
1126
1127        if withdraw_occurred {
1128            self.withdrawn_amount += pool.total_unstaked.0;
1129            // if a withdraw occurred, the new total unstake amount on the pool should be the
                    amount
1130            // requested in this unstake.
1131            pool.total_unstaked = amount;
1132        } else {
1133            // if no withdraw occurred we add the requested unstake amount to the pool total
                    unstaked amount
1134            pool.total_unstaked = (pool.total_unstaked.0 + amount.0).into();
1135        }
1136
1137
1138        // update delegation pool and total_staked
1139        pool.last_unstake = Some(env::epoch_height());
1140        pool.total_staked = (pool.total_staked.0 - amount.0).into();
1141        self.total_staked -= amount.0;
1142        self.tax_exempt_stake = self.tax_exempt_stake.saturating_sub(amount.0);
1143        log!("Updated total_staked: {}", self.total_staked);
1144
1145
1146        log!(
```

```
1147            "New unstaked amount {}. Pool total unstaked {}. Was withdrawn: {}. Pre balance {}.
                    Post balance {}",
1148            new_unstaked_amount,
1149            pool.total_unstaked.0,
1150            withdraw_occurred,
1151            pre_unstake_staker_balance,
1152            env::account_balance()
1153        );
1154
1155
1156        // create the unstake request
1157        self.unstake_nonce += 1;
1158
1159
1160        let unstake_request = UnstakeRequest {
1161            pool_id: pool_id.clone(),
1162            near_amount: amount.0,
1163            user: caller.clone(),
1164            epoch: env::epoch_height(),
1165        };
1166
1167
1168        self.unstake_requests
1169            .insert(self.unstake_nonce, unstake_request);
1170
1171
1172        // refund any excess NEAR to allocator
1173        let storage_cost = NearToken::from_yoctonear(Self::get_storage_cost().0);
1174        if attached_near > storage_cost {
1175            Promise::new(caller.clone()).transfer(attached_near.checked_sub(storage_cost).unwrap())
                    ;
1176        }
1177
1178
1179        // emit Unstaked event
1180        Event::UnstakedEvent {
1181            user_id: &caller,
1182            amount: &amount,
1183            user_balance: &U128(self.token.accounts.get(&caller).unwrap_or(0)),
1184            shares_amount: &shares_amount,
1185            total_staked: &U128(self.total_staked),
1186            total_supply: &U128(self.token.total_supply),
1187            share_price_num: &share_price_num,
1188            share_price_denom: &share_price_denom,
1189            unstake_nonce: &U128(self.unstake_nonce),
1190            epoch: &env::epoch_height().into(),
1191            pool_id: &pool_id,
1192        }
1193        .emit();
1194    }
```

Listing 2.8: lib.rs

```
651 /// Allocates NEAR staking rewards to a recipient. Requires a storage deposit for new
        allocations
652 /// that is refunded upon deallocation.
653 #[payable]
654 pub fn allocate(&mut self, recipient: AccountId, amount: U128) {
655     self.check_not_paused();
656     self.check_whitelisted();
657     let allocator = env::predecessor_account_id();
658     let amount = amount.0;
659
660
661     require!(recipient != allocator, ERR_INVALID_RECIPIENT);
662     require!(amount >= ONE_NEAR, ERR_ALLOCATION_UNDER_ONE_NEAR);
663
664
665     let (global_share_price_num, global_share_price_denom) = Self::internal_share_price(
666         self.total_staked,
667         self.token.ft_total_supply().0,
668         self.tax_exempt_stake,
669         self.fee,
670     );
671
672
673     let mut storage_cost = NearToken::from_near(0);
674     let attached_deposit = env::attached_deposit();
675
676
677     let allocation = self
678         .allocations
679         .entry(allocator.clone())
680         .or_default() // fetches the users allocations or creates a new hashmap
681         .entry(recipient.clone()) //fetches the allocation to the recipient
682         .and_modify(|allocation| {
683             //updates the recipients allocation if it exists
684             *allocation = Self::calculate_updated_allocation(
685                 allocation,
686                 amount,
687                 global_share_price_num,
688                 global_share_price_denom,
689             )
690         })
691         .or_insert_with(|| {
692             storage_cost = NearToken::from_yoctonear(Self::get_storage_cost().0);
693             if attached_deposit < storage_cost {
694                 env::panic_str(ERR_STORAGE_DEPOSIT_TOO_SMALL);
695             }
696             Allocation {
697                 // inserts a new allocation if one doesn't exist for this recipient
698                 near_amount: amount,
699                 share_price_num: global_share_price_num,
700                 share_price_denom: global_share_price_denom,
701             }
702         });
```

```
703
704
705         let updated_allocation = *allocation;
706         let (
707             total_allocated_amount,
708             total_allocated_share_price_num,
709             total_allocated_share_price_denom,
710         ) = self.get_total_allocated(allocator.clone());
711
712
713         // refund any excess NEAR to allocator
714         if attached_deposit > storage_cost {
715             Promise::new(allocator.clone())
716                 .transfer(attached_deposit.checked_sub(storage_cost).unwrap());
717         }
718
719
720         // emit event
721         Event::AllocatedEvent {
722             user: &allocator,
723             recipient: &recipient,
724             amount: &amount.into(),
725             total_amount: &updated_allocation.near_amount.into(),
726             share_price_num: &updated_allocation.share_price_num.to_string(),
727             share_price_denom: &updated_allocation.share_price_denom.to_string(),
728             total_allocated_amount: &total_allocated_amount,
729             total_allocated_share_price_num: &total_allocated_share_price_num,
730             total_allocated_share_price_denom: &total_allocated_share_price_denom,
731         }
732         .emit();
733     }
```

Listing 2.9: lib.rs

Impact The accuracy of operations such as `allocate()` and `distribute_reward()` would be affected.

Suggestion The operation to burn shares should be placed within the callback function `finalize_unstake()`, and it should be ensured that the above operations execute `check_contract_in_sync()` to make sure the share price is updated.

2.1.5 Lack of callback function `ft_resolve_transfer()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `TruNEAR` token contract does not implement the function `ft_resolve_transfer()`. This causes `ft_transfer_call()` to error out when invoking the receiver's `ft_on_transfer()` and handling the corresponding result. However, since it is a cross-contract call, the token transfer operation inside the token contract will not revert. In this scenario, if the receiver

contract rejects the transfer by not recording it in `ft_on_transfer()`, the token contract would still record the token under the receiver contract, meaning the receiver contract would hold the token that it cannot withdraw, leading to a consistency issue.

```
#[near]
impl FungibleTokenCore for NearStaker {
    /// Sends TruNEAR to another registered account.
    #[payable]
    fn ft_transfer(&mut self, receiver_id: AccountId, amount: U128, memo: Option<String>) {
        self.token.ft_transfer(receiver_id, amount, memo)
    }


    /// Transfers with a callback to the receiver contract.
    #[payable]
    fn ft_transfer_call(
        &mut self,
        receiver_id: AccountId,
        amount: U128,
        memo: Option<String>,
        msg: String,
    ) -> PromiseOrValue<U128> {
        self.token.ft_transfer_call(receiver_id, amount, memo, msg)
    }


    /// Returns the total supply of the token.
    fn ft_total_supply(&self) -> U128 {
        self.token.ft_total_supply()
    }


    /// Returns the balance of the account. If the account doesn't exist it returns `"0"`.
    fn ft_balance_of(&self, account_id: AccountId) -> U128 {
        self.token.ft_balance_of(account_id)
    }
}
```

Listing 2.10: core.rs

Impact The token transfer may succeed at the token contract level but fail at the receiver contract level if `ft_on_transfer()` rejects it.

Suggestion Implement the function `ft_resolve_transfer()` accordingly.

2.1.6 Lack of timely account registration for treasury in function `set_treasury()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The contract `owner` can change the `treasury` via the function `set_treasury()`, where the `treasury` acts as the fee recipient. However, after updating the `treasury`, the account

is not promptly registered. Specifically, in the function `distribute_reward()`, a portion of the reward is sent as a fee to the `treasury` in the form of shares. If the treasury is not registered, the function `internal_transfer()` fails, preventing `distribute_reward()` from executing properly.

```
405   pub fn set_treasury(&mut self, new_treasury: AccountId) {
406       self.check_owner();
407       Event::SetTreasuryEvent {
408           old_treasury: &self.treasury,
409           new_treasury: &new_treasury,
410       }
411       .emit();
412       self.treasury = new_treasury;
413   }
```

Listing 2.11: lib.rs

```
785   pub fn distribute_rewards(&mut self, recipient: AccountId, in_near: bool) {
786       self.check_not_paused();
787       self.check_whitelisted();
788
789
790       let distributor = env::predecessor_account_id();
791
792
793       let user_allocations = self
794           .allocations
795           .get(&distributor)
796           .expect(ERR_NO_ALLOCATIONS);
797
798
799       require!(
800           user_allocations.contains_key(&recipient),
801           ERR_NO_ALLOCATIONS_TO_RECIPIENT
802       );
803
804
805       let (global_price_num, global_price_denom) = Self::internal_share_price(
806           self.total_staked,
807           self.token.total_supply,
808           self.tax_exempt_stake,
809           self.fee,
810       );
811       let attached_near = env::attached_deposit();
812
813
814       let distribution_info_result = self.internal_distribute(
815           distributor.clone(),
816           recipient.clone(),
817           global_price_num,
818           global_price_denom,
819           in_near,
820           attached_near,
821       );
```

```
822
823
824         match distribution_info_result {
825             Err(error) => {
826                 env::panic_str(error.to_string().as_str());
827             }
828             Ok(distribution_info_opt) => {
829                 if distribution_info_opt.is_none() {
830                     log!("No rewards to distribute");
831                     if attached_near.as_yoctonear() > 0 {
832                         Promise::new(distributor.clone()).transfer(attached_near);
833                     }
834                     return;
835                 }
836
837
838                 // refund any excess NEAR to distributor
839                 let distribution_info = distribution_info_opt.unwrap();
840                 if distribution_info.refund_amount > 0 {
841                     Promise::new(distributor.clone())
842                         .transfer(NearToken::from_yoctonear(distribution_info.refund_amount));
843                 }
844
845
846                 let (
847                     total_allocated_amount,
848                     total_allocated_share_price_num,
849                     total_allocated_share_price_denom,
850                 ) = self.get_total_allocated(distributor.clone());
851
852
853                 // emit Distribute Rewards event
854                 Event::DistributedRewardsEvent {
855                     user: distributor.clone(),
856                     recipient: recipient.clone(),
857                     shares: U128(distribution_info.shares_amount),
858                     near_amount: U128(distribution_info.near_amount),
859                     user_balance: self.ft_balance_of(distributor),
860                     recipient_balance: self.ft_balance_of(recipient),
861                     fees: distribution_info.fees.into(),
862                     treasury_balance: self.ft_balance_of(self.treasury.clone()),
863                     share_price_num: distribution_info.share_price_num.to_string(),
864                     share_price_denom: distribution_info.share_price_denom.to_string(),
865                     in_near,
866                     total_allocated_amount,
867                     total_allocated_share_price_num,
868                     total_allocated_share_price_denom,
869                 }
870                 .emit();
871             }
872         }
873     }
```

Listing 2.12: lib.rs

```rust
pub(crate) fn internal_distribute(
    &mut self,
    distributor: AccountId,
    recipient: AccountId,
    global_price_num: U256,
    global_price_denom: U256,
    in_near: bool,
    attached_near: NearToken,
) -> Result<Option<DistributionInfo>, Box<dyn std::error::Error>> {
    let allocation = self
        .allocations
        .get_mut(&distributor)
        .unwrap()
        .get_mut(&recipient)
        .unwrap();


    // check if there are any rewards to distribute. If not, return no distribution
    if allocation.share_price_num / allocation.share_price_denom
        == global_price_num / global_price_denom
    {
        return Ok(None);
    }


    // calculate the amount of rewards that have accumulated
    let mut shares_to_move = Self::internal_calculate_distribution_amount(
        allocation,
        global_price_num,
        global_price_denom,
    );


    // calculate the distribution fee if applicable
    let fees = shares_to_move * (self.distribution_fee as u128) / (FEE_PRECISION as u128);


    if fees > 0 {
        shares_to_move -= fees;
        self.token
            .internal_transfer(&distributor, &self.treasury, fees, None);
    }


    // calculate the amount of rewards in NEAR
    let near_amount = NearToken::from_yoctonear(Self::convert_to_assets(
        shares_to_move,
        global_price_num,
        global_price_denom,
        false,
```

```
466         ));
467
468
469         let refund_amount;
470         if in_near {
471             if near_amount > attached_near {
472                 // if don't have enough NEAR return an error
473                 return Err(ERR_INSUFFICIENT_NEAR_BALANCE.into());
474             };
475
476
477             // calculate the refund amount and transfer NEAR to the recipient
478             refund_amount = attached_near.checked_sub(near_amount).unwrap();
479             Promise::new(recipient.clone()).transfer(near_amount);
480         } else {
481             let distributor_trunear_balance = self.token.ft_balance_of(distributor.clone()).0;
482             if shares_to_move > distributor_trunear_balance {
483                 // if don't have enough TruNEAR return an error
484                 return Err(ERR_INSUFFICIENT_TRUNEAR_BALANCE.into());
485             };
486
487
488             // register the recipient if they don't have a TruNEAR account
489             if !self.token.accounts.contains_key(&recipient) {
490                 self.token.accounts.insert(&recipient, &0);
491             }
492
493
494             // transfer TruNEAR to the recipient and refund any attached NEAR
495             refund_amount = attached_near;
496             self.token
497                 .internal_transfer(&distributor, &recipient, shares_to_move, None);
498         }
499
500
501         // update the allocation and return the distribution info
502         allocation.share_price_num = global_price_num;
503         allocation.share_price_denom = global_price_denom;
504
505
506         Ok(Some(DistributionInfo {
507             shares_amount: shares_to_move,
508             near_amount: near_amount.as_yoctonear(),
509             refund_amount: refund_amount.as_yoctonear(),
510             fees,
511             share_price_num: allocation.share_price_num,
512             share_price_denom: allocation.share_price_denom,
513         }))
514     }
```

Listing 2.13: lib.rs

```
97  pub fn internal_transfer(
```

```
98      &mut self,
99      sender_id: &AccountId,
100     receiver_id: &AccountId,
101     amount: Balance,
102     memo: Option<String>,
103 ) {
104     require!(
105         sender_id != receiver_id,
106         "Sender and receiver should be different"
107     );
108     require!(amount > 0, "The amount should be a positive number");
109     self.internal_withdraw(sender_id, amount);
110     self.internal_deposit(receiver_id, amount);
111     FtTransfer {
112         old_owner_id: sender_id,
113         new_owner_id: receiver_id,
114         amount: U128(amount),
115         memo: memo.as_deref(),
116     }
117     .emit();
118 }
```

Listing 2.14: core_impl.rs

```
71  pub fn internal_deposit(&mut self, account_id: &AccountId, amount: Balance) {
72      let balance = self.internal_unwrap_balance_of(account_id);
73      if let Some(new_balance) = balance.checked_add(amount) {
74          self.accounts.insert(account_id, &new_balance);
75          self.total_supply = self
76              .total_supply
77              .checked_add(amount)
78              .unwrap_or_else(|| env::panic_str(ERR_TOTAL_SUPPLY_OVERFLOW));
79      } else {
80          env::panic_str("Balance overflow");
81      }
82  }
```

Listing 2.15: core_impl.rs

Impact The function `distribute_rewards()` cannot execute properly.

Suggestion Revise the logic to ensure that the `treasury` is promptly registered in the token system after it is updated.

2.1.7 Potential unrefunded fee in function `distribute_all()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `lib.rs`, the function `distribute_all()` is used to distribute rewards to all recipients from their distributor. The distributor can choose to either distribute `NEAR` or

TruNEAR as rewards. If the distributor chooses NEAR, the function get_rewards_distribution_amounts() will be executed to return the required amounts of fees (TruNEAR) and NEAR. Specifically, it first calculates the total amount of rewards to be distributed and then computes the fee proportionally, rounding down only once. This leads to the calculated required_near being slightly underestimated and required_shares being overestimated, which passes the check that ensures that the distributor's attached NEAR is greater than or equal to required_near. However, in the function internal_distribute(), the rewards for each recipient are iterated over, and the fee is calculated during each iteration. Due to multiple rounds of rounding down across several iterations, the total spent on fees ends up being less than required_shares, and the total NEAR spent exceeds required_near. Therefore, the function reaches the branch at line 464, where the distributor fails to successfully distribute the rewards, and the paid fees are not refunded, which is incorrect.

```
868   pub fn distribute_all(&mut self, in_near: bool) {
869       self.check_not_paused();
870       self.check_whitelisted();
871
872
873       // check if distributor has allocations
874       let distributor = env::predecessor_account_id();
875       require!(
876           self.allocations.contains_key(&distributor),
877           ERR_NO_ALLOCATIONS
878       );
879
880
881       // ensure distributor has enough NEAR and TruNEAR to complete the distribution
882       let (required_shares, required_near) =
883           self.get_rewards_distribution_amounts(&distributor, None, in_near);
884       if self.ft_balance_of(distributor.clone()).0 < required_shares.0 {
885           env::panic_str(ERR_INSUFFICIENT_TRUNEAR_BALANCE);
886       }
887       if env::attached_deposit().as_yoctonear() < required_near.0 {
888           env::panic_str(ERR_INSUFFICIENT_NEAR_BALANCE);
889       }
890
891
892       let (total_allocated_amount, _, _) = self.get_total_allocated(distributor.clone());
893
894
895       let (global_price_num, global_price_denom) = Self::internal_share_price(
896           self.total_staked,
897           self.token.total_supply,
898           self.tax_exempt_stake,
899           self.fee,
900       );
901
902
903       // this holds the amount of NEAR we will need to refund to the distributor at the end of
              the distribution
904       let mut refund_near_amount = env::attached_deposit();
```

```
905
906
907         let distributor_allocations = self.allocations.get(&distributor).cloned().unwrap();
908
909
910         let mut distributed_rewards_events: Vec<Event> = vec![];
911
912
913         distributor_allocations.keys().for_each(|recipient| {
914             let distribution_info_result = self.internal_distribute(
915                 distributor.clone(),
916                 recipient.clone(),
917                 global_price_num,
918                 global_price_denom,
919                 in_near,
920                 refund_near_amount,
921             );
922
923
924             match distribution_info_result {
925                 Err(error) => {
926                     log!("Error distributing rewards: {}", error);
927                 }
928                 Ok(distribution_info_opt) => {
929                     match distribution_info_opt {
930                         None => log!("No rewards to distribute to {}", recipient),
931                         Some(distribution_info) => {
932                             // update the near amount left for the next distribution
933                             refund_near_amount =
934                                 NearToken::from_yoctonear(distribution_info.refund_amount);
935                             distributed_rewards_events.push(Event::DistributedRewardsEvent {
936                                 user: distributor.clone(),
937                                 recipient: recipient.clone(),
938                                 shares: U128(distribution_info.shares_amount),
939                                 near_amount: U128(distribution_info.near_amount),
940                                 user_balance: self.ft_balance_of(distributor.clone()),
941                                 recipient_balance: self.ft_balance_of(recipient.clone()),
942                                 fees: distribution_info.fees.into(),
943                                 treasury_balance: self.ft_balance_of(self.treasury.clone()),
944                                 share_price_num: distribution_info.share_price_num.to_string(),
945                                 share_price_denom: distribution_info.share_price_denom.to_string(),
946                                 in_near,
947                                 total_allocated_amount,
948                                 total_allocated_share_price_num: global_price_num.to_string(),
949                                 total_allocated_share_price_denom: global_price_denom.to_string(),
950                             });
951                         }
952                     };
953                 }
954             }
955         });
956
957
```

```
958         // refund any excess NEAR to distributor
959         if refund_near_amount.as_yoctonear() > 0 {
960             Promise::new(distributor.clone()).transfer(refund_near_amount);
961         }
962
963
964         // emit DistributedRewardsEvent events
965         distributed_rewards_events
966             .iter()
967             .for_each(|event| event.emit());
968
969
970         // emit DistributedAllEvent
971         Event::DistributedAllEvent { user: &distributor }.emit();
972     }
```

Listing 2.16: lib.rs

```
259     pub fn get_rewards_distribution_amounts(
260         &self,
261         distributor: &AccountId,
262         recipient: Option<AccountId>,
263         in_near: bool,
264     ) -> (U128, U128) {
265         let user_allocations = self.allocations.get(distributor);
266
267
268         // if the distributor has no allocations no TruNEAR or NEAR is needed
269         if user_allocations.is_none() {
270             return (U128(0), U128(0));
271         };
272
273
274         let (global_price_num, global_price_denom) = Self::internal_share_price(
275             self.total_staked,
276             self.ft_total_supply().0,
277             self.tax_exempt_stake,
278             self.fee,
279         );
280
281
282         let required_shares;
283         if let Some(r) = recipient {
284             // calculate the amount of shares required to distribute to a single recipient
285             let allocation = user_allocations
286                 .unwrap()
287                 .get(&r)
288                 .expect(ERR_NO_ALLOCATIONS_TO_RECIPIENT);
289
290
291             required_shares = Self::internal_calculate_distribution_amount(
292                 allocation,
293                 global_price_num,
```

```
294             global_price_denom,
295         );
296     } else {
297         // calculate the amount of shares required to distribute to all recipients
298         required_shares = user_allocations
299             .unwrap()
300             .iter()
301             .map(|(_, allocation)| allocation)
302             .fold(0, |acc, a| {
303                 acc + Self::internal_calculate_distribution_amount(
304                     a,
305                     global_price_num,
306                     global_price_denom,
307                 )
308             });
309     }
310
311
312     if in_near {
313         // for NEAR distributions fees are deducted from the required NEAR amount and accounted
                as required TruNEAR
314         let fees = required_shares * (self.distribution_fee as u128) / (FEE_PRECISION as u128);
315         let required_near = Self::convert_to_assets(
316             required_shares - fees,
317             global_price_num,
318             global_price_denom,
319             false,
320         );
321         (U128::from(fees), U128::from(required_near))
322     } else {
323         // for TruNEAR distributions the required TruNEAR amount includes the distribution fees
324         (U128::from(required_shares), U128(0))
325     }
326 }
```

Listing 2.17: lib.rs

```
416 pub(crate) fn internal_distribute(
417     &mut self,
418     distributor: AccountId,
419     recipient: AccountId,
420     global_price_num: U256,
421     global_price_denom: U256,
422     in_near: bool,
423     attached_near: NearToken,
424 ) -> Result<Option<DistributionInfo>, Box<dyn std::error::Error>> {
425     let allocation = self
426         .allocations
427         .get_mut(&distributor)
428         .unwrap()
429         .get_mut(&recipient)
430         .unwrap();
431
```

```
432
433         // check if there are any rewards to distribute. If not, return no distribution
434         if allocation.share_price_num / allocation.share_price_denom
435             == global_price_num / global_price_denom
436         {
437             return Ok(None);
438         }
439
440
441         // calculate the amount of rewards that have accumulated
442         let mut shares_to_move = Self::internal_calculate_distribution_amount(
443             allocation,
444             global_price_num,
445             global_price_denom,
446         );
447
448
449         // calculate the distribution fee if applicable
450         let fees = shares_to_move * (self.distribution_fee as u128) / (FEE_PRECISION as u128);
451
452
453         if fees > 0 {
454             shares_to_move -= fees;
455             self.token
456                 .internal_transfer(&distributor, &self.treasury, fees, None);
457         }
458
459
460         // calculate the amount of rewards in NEAR
461         let near_amount = NearToken::from_yoctonear(Self::convert_to_assets(
462             shares_to_move,
463             global_price_num,
464             global_price_denom,
465             false,
466         ));
467
468
469         let refund_amount;
470         if in_near {
471             if near_amount > attached_near {
472                 // if don't have enough NEAR return an error
473                 return Err(ERR_INSUFFICIENT_NEAR_BALANCE.into());
474             };
475
476
477             // calculate the refund amount and transfer NEAR to the recipient
478             refund_amount = attached_near.checked_sub(near_amount).unwrap();
479             Promise::new(recipient.clone()).transfer(near_amount);
480         } else {
481             let distributor_trunear_balance = self.token.ft_balance_of(distributor.clone()).0;
482             if shares_to_move > distributor_trunear_balance {
483                 // if don't have enough TruNEAR return an error
484                 return Err(ERR_INSUFFICIENT_TRUNEAR_BALANCE.into());
```

```
485             };
486
487
488             // register the recipient if they don't have a TruNEAR account
489             if !self.token.accounts.contains_key(&recipient) {
490                 self.token.accounts.insert(&recipient, &0);
491             }
492
493
494             // transfer TruNEAR to the recipient and refund any attached NEAR
495             refund_amount = attached_near;
496             self.token
497                 .internal_transfer(&distributor, &recipient, shares_to_move, None);
498         }
499
500
501         // update the allocation and return the distribution info
502         allocation.share_price_num = global_price_num;
503         allocation.share_price_denom = global_price_denom;
504
505
506         Ok(Some(DistributionInfo {
507             shares_amount: shares_to_move,
508             near_amount: near_amount.as_yoctonear(),
509             refund_amount: refund_amount.as_yoctonear(),
510             fees,
511             share_price_num: allocation.share_price_num,
512             share_price_denom: allocation.share_price_denom,
513         }))
514     }
```

Listing 2.18: internal.rs

Impact The distributor fails to distribute rewards to recipients with the fees not refunded.

Suggestion Revise the logic to ensure that fees are promptly refunded to the distributor if the rewards are not successfully distributed.

2.2 Additional Recommendation

2.2.1 Lack of check in function `check_owner()`

Status Confirmed

Introduced by `Version 1`

Description In the file `lib.rs`, there are multiple privileged methods such as `add_pool()`, `enable_pool()`, and `disable_pool()`. These functions only verify that the invoker is the contract owner but do not check that the attached `NEAR` equals `1 yocto NEAR`, which doesn't follow the best practice for implementing sensitive functions.

```
32     pub(crate) fn check_owner(&self) {
33         require!(
```

```
34          self.owner_id == env::predecessor_account_id(),
35          ERR_ONLY_OWNER
36      );
37  }
```

Listing 2.19: internal.rs

Suggestion Add checks in above functions ensure the attached `NEAR` equals `1 yocto NEAR`, and annotate these functions with `[payable]`.

Feedback from the project The `owner` is a multisig account.

2.2.2 Redundant code

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `internal_check_unstake_amount()`, line 516 is redundant as the function `max_withdraw()` has already been invoked earlier. The subsequent check should directly use `max_withdraw` without invoking it again.

```
507  pub(crate) fn internal_check_unstake_amount(
508      &self,
509      pool_id: &AccountId,
510      amount: u128,
511      caller: &AccountId,
512  ) -> u128 {
513      // check if user has enough TruNEAR to unstake
514      let max_withdraw = self.max_withdraw(caller.clone()).0;
515      require!(
516          self.max_withdraw(caller.clone()) >= U128(amount),
517          ERR_INVALID_UNSTAKE_AMOUNT
518      );
519
520
521      // if the user's remaining balance falls below one NEAR, unstake the entire user stake
522      let unstake_amount = if max_withdraw - amount < ONE_NEAR {
523          max_withdraw
524      } else {
525          amount
526      };
527
528
529      // check if there's enough staked balance to unstake on the pool
530      require!(
531          self.delegation_pools.get(pool_id).unwrap().total_staked >= U128(unstake_amount),
532          ERR_INSUFFICIENT_FUNDS_ON_POOL
533      );
534
535
536      unstake_amount
537  }
```

Listing 2.20: internal.rs

Suggestion Remove this redundant code.

2.2.3 Using constants in function `internal_deposit_and_stake()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `send_stake_promises()`, line 103 specifies the amount of gas attached for the first cross-contract invocation. Specifically, in the file `constants.rs`, `XCC_GAS` is defined as the gas paid for cross-contract invocations. It is recommended to replace the specified gas amount with `XCC_GAS` to ensure consistency throughout the code.

```
88  pub(crate) fn send_stake_promises(
89      pool_id: AccountId,
90      amount: u128,
91      caller: AccountId,
92  ) -> Promise {
93      let staker_id: AccountId = env::current_account_id();
94
95
96      let staker_arg = json!({ "account_id": staker_id }).to_string().into_bytes();
97
98
99      // we first call deposit_and_stake followed by get_account_total_balance to ensure the
            stake has been added
100     Promise::new(pool_id.clone())
101         .function_call(
102             "deposit_and_stake".to_owned(),
103             NO_ARGS,
104             NearToken::from_yoctonear(amount),
105             Gas::from_tgas(30),
106         )
107         .function_call(
108             "get_account_total_balance".to_owned(),
109             staker_arg,
110             NO_DEPOSIT,
111             XCC_GAS,
112         )
113         .then(
114             Self::ext(env::current_account_id())
115                 .with_static_gas(XCC_GAS)
116                 .finalize_deposit_and_stake(pool_id, U128(amount), caller),
117         )
118 }
```

Listing 2.21: internal.rs

```
8  pub const XCC_GAS: Gas = Gas::from_tgas(30); // approx gas needed for cross-contract calls
```

Listing 2.22: constant.rs

Suggestion Replace with `XCC_GAS`.

2.3 Note

2.3.1 Potential centralization risk

Introduced by `Version 1`

Description The protocol includes several privileged functions, such as `add_pool()`, and `enable_pool()`. If the `owner`'s private key is lost or maliciously exploited, it could potentially cause losses to users.

Feedback from the project The `owner` is a multisig account.

2.3.2 Potential discrepancies in share price calculations

Introduced by `Version 1`

Description Whitelisted users can deposit `NEAR` through the function `stake()`, which adds funds to the `staking_pool` and mints corresponding shares for the users. However, a potential situation arises if the deposit process spans into a new epoch, leading to potential discrepancies in share price calculations. Specifically, if a user invokes the function `stake()` during the last block of the previous epoch, the function `finalize_deposit_and_stake()` may be executed in the new epoch. Since rewards are generated each epoch, the actual share price in the new epoch might exceed the recorded value in the contract. This could result in the issuance of more shares than intended. A similar situation may also occur in the function `send_unstake_promises()`.

Feedback from the project When a user stakes `NEAR`, we establish a contract to provide them with a specific amount of `TruNEAR` in return. If the share price rises between the initiation of the transaction and the issuance of the liquid tokens, the user is still entitled to the originally promised quantity of tokens. Under no circumstances should the user receive fewer tokens than initially agreed upon due to an increase in share price.

2.3.3 Fixed token of distribution fee

Introduced by `Version 1`

Description In the reward distribution process, the distributor can choose to distribute rewards in `NEAR`, but they must pay the distribution fee using `TruNEAR`. This is inconsistent with the `in_near` flag.

Feedback from the project Indeed, the `in_near` flag specifies the token the user intends to distribute. Whether the treasury receives payment in `TruNEAR` or `NEAR` has no effect, adverse or not, on the distributor or recipient.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS