BLOCKSEC

Security Audit Report for rNEAR Contract

Date: August 5, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Rhea Finance
Target	rNEAR Contract

Version History

Version	Date	Description
1.0	August 5, 2025	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-08-05 11:58:06+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 [1] of rNEAR Contract of Rhea Finance.

The project is deployed on the NEAR network. Users can deposit NEAR into the protocol, which mints a corresponding amount of rNEAR based on the current share price. Conversely, users can also unstake by burning their rNEAR in exchange for the corresponding amount of NEAR. Users holding rNEAR receive PoS rewards.

Note this audit only focuses on the smart contracts in the following directories/files:

- contracts/lst/src

Other files are not within the scope of the audit. Additionally, all dependencies of the smart contracts within the audit scope are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

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
rNEAR Contract	Version 1	2fbd60eee96405e99b54b0b88774dd2c12017745
	Version 2	c7f4611f3886355e043c03157e7a33bdca8821e4

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 can-

[1] https://github.com/ref-finance/rnear-contract

not 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 Security Issues

* Access control
* Permission management
* Whitelist and blacklist mechanisms
* Initialization consistency
* Improper use of the proxy system
* Reentrancy
* Denial of Service (DoS)
* Untrusted external call and control flow
* Exception handling
* Data handling and flow
* Events operation
* Error-prone randomness
* Oracle security
* Business logic correctness
* Semantic and functional consistency
* Emergency mechanism
* Economic and incentive impact

1.3.2 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 five 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.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **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 found **five** potential security issues. Besides, we have **one** recommendation and **three** notes.

- High Risk: 3
- Low Risk: 2
- Recommendation: 1
- Note: 3

ID	Severity	Description	Category	Status
1	High	Incorrect logic in deposit operations	Security Issue	Fixed
2	High	Incorrect rewards distribution for beneficiary	Security Issue	Fixed
3	High	Validator pool lock due to unhandled callback and reward minting failure	Security Issue	Fixed
4	Low	Lack of executing status check in function `remove_validator()`	Security Issue	Fixed
5	Low	Improper external method invocation in function `drain_unstake()`	Security Issue	Confirmed
6	-	Lack of one yocto check in privileged functions	Recommendation	Fixed
7	-	Potential centralization risk	Note	-
8	-	Ensure timely invocation of function `epoch_update_rewards()`	Note	-
9	-	Potential arbitrage opportunity	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Incorrect logic in deposit operations

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Users can invoke `deposit_and_stake()` to deposit `NEAR` and receive `rNEAR` in return, based on the current share price. If it is the user's first deposit, a portion of `attached_deposit()` is deducted as the `storage_fee`. However, in the `storage_deposit()` function, if the attached deposit exceeds `storage_balance_bounds.min()`, the excess `NEAR` is refunded to the user. The protocol incorrectly proceeds to mint `rNEAR` shares based on the `attached_deposit()`, even though part of that amount has already been refunded. As a result, users receive `rNEAR` without actually staking the corresponding amount of `NEAR`, which is incorrect.

The same issue also exists in the function `deposit()`.

```
117    pub fn deposit_and_stake(&mut self) -> U128 {
118        let amount = env::attached_deposit().as_yoctonear();
```

```
119         let storage_used = if self.storage_balance_of(env::predecessor_account_id()).is_none() {
120             self.storage_deposit(None, None);
121             // log!("do register, use {}", self.storage_balance_bounds().min.as_yoctonear());
122             self.storage_balance_bounds().min.as_yoctonear()
123         } else {
124             // log!("already registered.");
125             0
126         };
127         self.internal_deposit(amount - storage_used);
128         self.internal_stake(amount - storage_used).into()
129     }
```

Listing 2.1: contracts/lst/src/stake_pool_itf.rs

```
 6   fn storage_deposit(
 7       &mut self,
 8       account_id: Option<AccountId>,
 9       registration_only: Option<bool>,
10   ) -> StorageBalance {
11       let acc_id = account_id.clone().unwrap_or(env::predecessor_account_id());
12       if self.data().accounts.get(&acc_id).is_none() {
13           self.data_mut()
14               .accounts
15               .insert(acc_id.clone(), Account::default());
16       }
17       self.data_mut().token.storage_deposit(account_id, registration_only)
18   }
```

Listing 2.2: contracts/lst/src/storage.rs

```
102   pub fn deposit(&mut self) {
103       let amount = env::attached_deposit().as_yoctonear();
104       let storage_used = if self.storage_balance_of(env::predecessor_account_id()).is_none() {
105           self.storage_deposit(None, None);
106           self.storage_balance_bounds().min.as_yoctonear()
107       } else {
108           0
109       };
110       self.internal_deposit(amount-storage_used);
111   }
```

Listing 2.3: contracts/lst/src/stake_pool_itf.rs

Impact Users receive `rNEAR` without actually staking the corresponding amount of `NEAR`.

Suggestion Revise the logic to ensure that the excess over the `storage_fee` is retained and properly applied to staking, rather than refunded.

2.1.2 Incorrect rewards distribution for beneficiary

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `epoch_update_rewards()` is responsible for updating the accumulated staking rewards. It first retrieves the `total_balance` of the validator from the staking pool and treats any excess over the locally recorded value as accumulated rewards. In the callback function `validator_get_balance_callback()`, the protocol first updates the share price, then distributes the rewards to each `beneficiary` based on a predefined allocation ratio. For each `beneficiary`, it calculates the amount of `NEAR` they are entitled to and mints the corresponding amount of `rNEAR` based on the updated share price.

However, because `rNEAR` is minted to each `beneficiary` individually, the `total_supply` of `rNEAR` increases incrementally during the process, causing the share price to gradually decrease. This means the earlier minted `rNEAR` represents less value than it should, which is incorrect.

```
201   pub fn epoch_update_rewards(&mut self, validator_id: AccountId) {
202       let min_gas = GAS_EPOCH_UPDATE_REWARDS.as_gas()
203           + GAS_EXT_GET_BALANCE.as_gas()
204           + GAS_CB_VALIDATOR_GET_BALANCE.as_gas();
205       require!(
206           env::prepaid_gas().as_gas() >= min_gas,
207           format!("{}. require at least {:?}", ERR_NO_ENOUGH_GAS, min_gas)
208       );
209
210       let mut validator = self
211           .data_mut()
212           .validator_pool
213           .get_validator(&validator_id)
214           .expect(ERR_VALIDATOR_NOT_EXIST);
215
216       validator
217           .refresh_total_balance(&mut self.data_mut().validator_pool)
218           .then(
219               Self::ext(env::current_account_id())
220                   .with_static_gas(GAS_CB_VALIDATOR_GET_BALANCE)
221                   .validator_get_balance_callback(validator.account_id),
222           );
223   }
```

Listing 2.4: contracts/lst/src/epoch_actions.rs

```
366   pub fn validator_get_balance_callback(
367       &mut self,
368       validator_id: AccountId,
369       #[callback] total_balance: U128,
370   ) {
371       let mut validator = self
372           .data_mut()
373           .validator_pool
374           .get_validator(&validator_id)
375           .expect(ERR_VALIDATOR_NOT_EXIST);
376
377       let new_balance = total_balance.0;
378       let rewards = new_balance - validator.total_balance();
```

```
379         Event::EpochUpdateRewards {
380             validator_id: &validator_id,
381             old_balance: &U128(validator.total_balance()),
382             new_balance: &U128(new_balance),
383             rewards: &U128(rewards),
384         }
385         .emit();
386
387         validator.on_new_total_balance(&mut self.data_mut().validator_pool, new_balance);
388
389         if rewards == 0 {
390             return;
391         }
392
393         self.data_mut().total_staked_near_amount += rewards;
394         self.internal_distribute_staking_rewards(rewards);
395     }
```

Listing 2.5: contracts/lst/src/epoch_actions.rs

```
22  pub(crate) fn internal_distribute_staking_rewards(&mut self, rewards: u128) {
23      let hashmap: HashMap<AccountId, u32> = self.internal_get_beneficiaries();
24      for (account_id, bps) in hashmap.iter() {
25          let reward_near_amount: u128 = bps_mul(rewards, *bps);
26          // mint extra LST for him
27          self.internal_mint_beneficiary_rewards(account_id, reward_near_amount);
28      }
29  }
30
31  /// Mint new LST tokens to given account at the current price.
32  /// This will DECREASE the LST price.
33  #[pause]
34  fn internal_mint_beneficiary_rewards(
35      &mut self,
36      account_id: &AccountId,
37      near_amount: u128,
38  ) -> ShareBalance {
39      let shares = self.num_shares_from_staked_amount_rounded_down(near_amount);
40      self.mint_lst(account_id, shares, Some("beneficiary rewards"));
41      shares
42  }
```

Listing 2.6: contracts/lst/src/internal.rs

Impact `Beneficiaries` incur a loss.

Suggestion Revise the logic to ensure the share price remains constant throughout the minting process.

2.1.3 Validator pool lock due to unhandled callback and reward minting failure

Severity High

Status Fixed in `Version 2`

Introduced by Version 1

Description The `epoch_update_rewards()` function initiates a cross-contract call to query staking rewards from the validator pool. Before the call, the validator's status is set to `executing` via function `pre_execution()` to prevent reentrant operations. Once the external call returns, the `validator_get_balance_callback()` is invoked to process the result.

This callback calculates the staking rewards and attempts to mint corresponding `rNEAR` tokens for distribution. However, if the reward amount is zero, the minting function will revert, which in turn causes the entire callback to fail. Because the validator's status is only reset to `executing = false` within the callback, any panic in this phase leaves the validator pool permanently locked in the executing state, making it impossible to perform future operations for that validator.

Furthermore, the callback function lacks proper handling of the cross-contract call result. It assumes success and proceeds directly with processing. In case of failure (e.g., target contract not responding), the validator will similarly remain locked in the executing state.

```
201   pub fn epoch_update_rewards(&mut self, validator_id: AccountId) {
202       let min_gas = GAS_EPOCH_UPDATE_REWARDS.as_gas()
203           + GAS_EXT_GET_BALANCE.as_gas()
204           + GAS_CB_VALIDATOR_GET_BALANCE.as_gas();
205       require!(
206           env::prepaid_gas().as_gas() >= min_gas,
207           format!("{}. require at least {:?}", ERR_NO_ENOUGH_GAS, min_gas)
208       );
209
210       let mut validator = self
211           .data_mut()
212           .validator_pool
213           .get_validator(&validator_id)
214           .expect(ERR_VALIDATOR_NOT_EXIST);
215
216       validator
217           .refresh_total_balance(&mut self.data_mut().validator_pool)
218           .then(
219               Self::ext(env::current_account_id())
220                   .with_static_gas(GAS_CB_VALIDATOR_GET_BALANCE)
221                   .validator_get_balance_callback(validator.account_id),
222           );
223   }
```

Listing 2.7: contracts/lst/src/epoch_actions.rs

```
169   pub fn refresh_total_balance(&mut self, pool: &mut ValidatorPool) -> Promise {
170       self.pre_execution(pool);
171
172       ext_staking_pool::ext(self.account_id.clone())
173           .with_static_gas(GAS_EXT_GET_BALANCE)
174           .get_account_total_balance(env::current_account_id())
175   }
```

Listing 2.8: contracts/lst/src/validator.rs

```rust
pub fn validator_get_balance_callback(
    &mut self,
    validator_id: AccountId,
    #[callback] total_balance: U128,
) {
    let mut validator = self
        .data_mut()
        .validator_pool
        .get_validator(&validator_id)
        .expect(ERR_VALIDATOR_NOT_EXIST);

    let new_balance = total_balance.0;
    let rewards = new_balance - validator.total_balance();
    Event::EpochUpdateRewards {
        validator_id: &validator_id,
        old_balance: &U128(validator.total_balance()),
        new_balance: &U128(new_balance),
        rewards: &U128(rewards),
    }
    .emit();

    validator.on_new_total_balance(&mut self.data_mut().validator_pool, new_balance);

    if rewards == 0 {
        return;
    }

    self.data_mut().total_staked_near_amount += rewards;
    self.internal_distribute_staking_rewards(rewards);
}
```

Listing 2.9: contracts/lst/src/epoch_actions.rs

```rust
pub fn on_new_total_balance(&mut self, pool: &mut ValidatorPool, new_total_balance: u128) {
    self.post_execution(pool);

    // sync base stake amount
    self.sync_base_stake_amount(pool, new_total_balance);
    // update staked amount
    self.staked_amount = new_total_balance - self.unstaked_amount;
    pool.save_validator(self);
}
```

Listing 2.10: contracts/lst/src/validator.rs

```rust
pub(crate) fn internal_distribute_staking_rewards(&mut self, rewards: u128) {
    let hashmap: HashMap<AccountId, u32> = self.internal_get_beneficiaries();
    for (account_id, bps) in hashmap.iter() {
        let reward_near_amount: u128 = bps_mul(rewards, *bps);
        // mint extra LST for him
        self.internal_mint_beneficiary_rewards(account_id, reward_near_amount);
    }
}
```

```rust

/// Mint new LST tokens to given account at the current price.
/// This will DECREASE the LST price.
#[pause]
fn internal_mint_beneficiary_rewards(
    &mut self,
    account_id: &AccountId,
    near_amount: u128,
) -> ShareBalance {
    let shares = self.num_shares_from_staked_amount_rounded_down(near_amount);
    self.mint_lst(account_id, shares, Some("beneficiary rewards"));
    shares
}
```

Listing 2.11: contracts/lst/src/internal.rs

```rust
pub fn mint_lst(&mut self, account_id: &AccountId, shares: u128, memo: Option<&str>) {
    require!(shares > 0, ERR_NON_POSITIVE_SHARES);
    // mint to account
    if self.data().token.accounts.get(account_id).is_none() {
        self.data_mut().token.internal_register_account(account_id);
    }
    self.data_mut().token.internal_deposit(account_id, shares);
    FtMint {
        owner_id: account_id,
        amount: U128(shares),
        memo,
    }
    .emit();
}
```

Listing 2.12: contracts/lst/src/lib.rs

Impact The validator pool may become permanently locked, blocking all subsequent operations for the affected validator.

Suggestion Add a check for cross-contract call success in the callback and ensure the executing flag is cleared even on failure or zero rewards.

2.1.4 Lack of executing status check in function `remove_validator()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the protocol, accounts with `OpManager` or `DAO` privileges can invoke the `remove_validator()` function to remove a specified `validator`. However, this function does not verify whether the `executing` flag of the `validator` is set to `true`.

Due to the asynchronous nature of the `NEAR` runtime, it is possible for functions `epoch_stake()` and `remove_validator()` to be invoked concurrently. In such a case, function `epoch_stake()`

may select a validator for staking, and before the cross-contract call resolves, function `remove_validator()` removes the validator from the pool. As a result, when the function `validator_staked_callback()` is triggered, it fails to find the validator in the pool and panics with `ERR_VALIDATOR_NOT_EXIST`, reverting user-related logic and stake settlement.

```
400   pub fn remove_validator(&mut self, validator_id: AccountId) -> Validator {
401       self.data_mut()
402           .validator_pool
403           .remove_validator(&validator_id)
404   }
```

Listing 2.13: contracts/lst/src/validator_pool.rs

```
314   pub fn remove_validator(&mut self, validator_id: &AccountId) -> Validator {
315       let validator: Validator = self
316           .validators
317           .remove(validator_id)
318           .expect(ERR_VALIDATOR_NOT_EXIST)
319           .into();
320
321       // make sure this validator is not used at all
322       require!(
323           validator.staked_amount == 0 && validator.unstaked_amount == 0,
324           ERR_VALIDATOR_IN_USE
325       );
326
327       self.total_weight -= validator.weight;
328       self.total_base_stake_amount -= validator.base_stake_amount;
329
330       Event::ValidatorRemoved {
331           account_id: validator_id,
332       }
333       .emit();
334
335       validator
336   }
```

Listing 2.14: contracts/lst/src/validator_pool.rs

```
52    pub fn epoch_stake(&mut self) -> PromiseOrValue<bool> {
53        // make sure enough gas was given
54        let min_gas = GAS_EPOCH_STAKE.as_gas()
55            + GAS_EXT_DEPOSIT_AND_STAKE.as_gas()
56            + GAS_CB_VALIDATOR_STAKED.as_gas()
57            + GAS_SYNC_BALANCE.as_gas()
58            + GAS_CB_VALIDATOR_SYNC_BALANCE.as_gas();
59        require!(
60            env::prepaid_gas().as_gas() >= min_gas,
61            format!("{}. require at least {:?}", ERR_NO_ENOUGH_GAS, min_gas)
62        );
63
64        self.epoch_cleanup();
65        // after cleanup, there might be no need to stake
```

```
66      if self.data().stake_amount_to_settle == 0 {
67          log!("no need to stake, amount to settle is zero");
68          return PromiseOrValue::Value(false);
69      }
70
71      let candidate = self.data().validator_pool.get_candidate_to_stake(
72          self.data().stake_amount_to_settle,
73          self.data().total_staked_near_amount,
74      );
75
76      if candidate.is_none() {
77          log!("no candidate found to stake");
78          return PromiseOrValue::Value(false);
79      }
80
81      let mut candidate = candidate.unwrap();
82      let amount_to_stake = candidate.amount;
83
84      if amount_to_stake < MIN_AMOUNT_TO_PERFORM_STAKE {
85          log!("stake amount too low: {}", amount_to_stake);
86          return PromiseOrValue::Value(false);
87      }
88
89      require!(
90          env::account_balance().as_yoctonear()
91              >= amount_to_stake + CONTRACT_MIN_RESERVE_BALANCE.as_yoctonear(),
92          ERR_MIN_RESERVE
93      );
94
95      // update internal state
96      self.data_mut().stake_amount_to_settle -= amount_to_stake;
97
98      Event::EpochStakeAttempt {
99          validator_id: &candidate.validator.account_id,
100         amount: &U128(amount_to_stake),
101     }
102     .emit();
103
104     // do staking on selected validator
105     candidate
106         .validator
107         .deposit_and_stake(&mut self.data_mut().validator_pool, amount_to_stake)
108         .then(
109             Self::ext(env::current_account_id())
110                 .with_static_gas(
111                     GAS_CB_VALIDATOR_STAKED
112                         .checked_add(GAS_SYNC_BALANCE)
113                         .unwrap()
114                         .checked_add(GAS_CB_VALIDATOR_SYNC_BALANCE)
115                         .unwrap(),
116                 )
117                 .validator_staked_callback(
118                     candidate.validator.account_id.clone(),
```

```
                    amount_to_stake.into(),
                ),
        )
        .into()
}
```

Listing 2.15: contracts/lst/src/epoch_actions.rs

```
pub fn validator_staked_callback(
    &mut self,
    validator_id: AccountId,
    amount: U128,
) -> PromiseOrValue<bool> {
    let amount = amount.into();
    let mut validator = self
        .data_mut()
        .validator_pool
        .get_validator(&validator_id)
        .unwrap_or_else(|| panic!("{}: {}", ERR_VALIDATOR_NOT_EXIST, &validator_id));

    if is_promise_success() {
        validator.on_stake_success(&mut self.data_mut().validator_pool, amount);

        Event::EpochStakeSuccess {
            validator_id: &validator_id,
            amount: &U128(amount),
        }
        .emit();

        validator
            .sync_account_balance(&mut self.data_mut().validator_pool, true)
            .then(
                Self::ext(env::current_account_id())
                    .with_static_gas(GAS_CB_VALIDATOR_SYNC_BALANCE)
                    .validator_get_account_callback(validator_id),
            )
            .into()
    } else {
        validator.on_stake_failed(&mut self.data_mut().validator_pool);

        // stake failed, revert
        self.data_mut().stake_amount_to_settle += amount;

        Event::EpochStakeFailed {
            validator_id: &validator_id,
            amount: &U128(amount),
        }
        .emit();

        PromiseOrValue::Value(false)
    }
}
```

Listing 2.16: contracts/lst/src/epoch_actions.rs

```
121   pub fn on_stake_success(&mut self, pool: &mut ValidatorPool, amount: u128) {
122       // Do not call post_execution() here because we need to sync account balance after stake
123       self.staked_amount += amount;
124       pool.save_validator(self);
125   }
```

Listing 2.17: contracts/lst/src/validator.rs

Impact Concurrent stake/removal may cause unexpected callback failure.

Suggestion Add a check to ensure the `validator`'s `executing` flag is `false` before allowing removal.

2.1.5 Improper external method invocation in function `drain_unstake()`

Severity Low

Status Confirmed

Introduced by `Version 1`

Description When the protocol decides to activate a `validator`, an account with `OpManager` or `DAO` privileges can invoke the function `drain_unstake()` to withdraw all assets from the protocol. However, the function incorrectly uses `unstake()` instead of `unstake_all()`. Specifically, the amount passed to function `unstake()` is the locally recorded `staked_amount`, which may be out of sync with the actual amount in the staking pool due to delayed reward updates. As a result, it may fail to withdraw all assets from the validator.

```
498   pub fn drain_unstake(&mut self, validator_id: AccountId) -> Promise {
499       // make sure enough gas was given
500       let min_gas = GAS_DRAIN_UNSTAKE.as_gas()
501           + GAS_EXT_UNSTAKE.as_gas()
502           + GAS_CB_VALIDATOR_UNSTAKED.as_gas()
503           + GAS_SYNC_BALANCE.as_gas()
504           + GAS_CB_VALIDATOR_SYNC_BALANCE.as_gas();
505       require!(
506           env::prepaid_gas().as_gas() >= min_gas,
507           format!("{}. require at least {:?}", ERR_NO_ENOUGH_GAS, min_gas)
508       );
509
510       let mut validator = self
511           .data_mut()
512           .validator_pool
513           .get_validator(&validator_id)
514           .expect(ERR_VALIDATOR_NOT_EXIST);
515
516       // make sure the validator:
517       // 1. has weight set to 0
518       // 2. has base stake amount set to 0
519       // 3. not in pending release
520       // 4. has not unstaked balance (because this part is from user's unstake request)
521       // 5. not in draining process
522       require!(validator.weight == 0, ERR_NON_ZERO_WEIGHT);
```

```
523         require!(
524             validator.base_stake_amount == 0,
525             ERR_NON_ZERO_BASE_STAKE_AMOUNT
526         );
527         require!(
528             !validator.pending_release(),
529             ERR_VALIDATOR_UNSTAKE_WHEN_LOCKED
530         );
531         // in practice we allow 1 NEAR due to the precision of stake operation
532         require!(
533             validator.unstaked_amount < ONE_NEAR,
534             ERR_BAD_UNSTAKED_AMOUNT
535         );
536         require!(!validator.draining, ERR_DRAINING);
537
538         let unstake_amount = validator.staked_amount;
539
540         Event::DrainUnstakeAttempt {
541             validator_id: &validator_id,
542             amount: &U128(unstake_amount),
543         }
544         .emit();
545
546         // perform actual unstake
547         validator
548             .unstake(&mut self.data_mut().validator_pool, unstake_amount)
549             .then(
550                 Self::ext(env::current_account_id())
551                     .with_static_gas(
552                         GAS_CB_VALIDATOR_UNSTAKED
553                             .checked_add(GAS_SYNC_BALANCE)
554                             .unwrap()
555                             .checked_add(GAS_CB_VALIDATOR_SYNC_BALANCE)
556                             .unwrap(),
557                     )
558                     .with_unused_gas_weight(0)
559                     .validator_drain_unstaked_callback(validator.account_id, unstake_amount.into()),
560             )
561     }
```

Listing 2.18: contracts/lst/src/validator_pool.rs

Impact The function `drain_unstake()` may fail to withdraw all assets from the `validator`.

Suggestion Replace the `unstake()` function with `unstake_all()`.

Feedback from the project The team promises that the function `drain_unstake()` is invoked only after the state has been synchronized.

2.2 Recommendation

2.2.1 Lack of one yocto check in privileged functions

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the protocol, several privileged functions such as `add_validator()` and `remove_validator()` do not enforce a check that the attached deposit is exactly one `yoctoNEAR`. This violates `NEAR`'s best security practices for contract development.

```
385    pub fn add_validator(&mut self, validator_id: AccountId, weight: u16) {
386        self.add_whitelisted_validator(&validator_id, weight);
387    }
```

Listing 2.19: contracts/lst/src/validator_pool.rs

```
400    pub fn remove_validator(&mut self, validator_id: AccountId) -> Validator {
401        self.data_mut()
402            .validator_pool
403            .remove_validator(&validator_id)
404    }
```

Listing 2.20: contracts/lst/src/validator_pool.rs

Suggestion Revise the logic to enforce that all privileged function calls attach exactly one `yoctoNEAR`.

2.3 Note

2.3.1 Potential centralization risk

Introduced by `Version 1`

Description In the current implementation, several privileged roles are set to govern and regulate the system-wide operations (e.g., parameter setting). Additionally, the `owner` also has the ability to upgrade the implementation. If the private keys of these privileged roles are lost or maliciously exploited, it could potentially lead to losses for users.

2.3.2 Ensure timely invocation of function `epoch_update_rewards()`

Introduced by `Version 1`

Description The function `epoch_update_rewards()` retrieves the `total_balance` of a specified `validator` from the staking pool and considers the amount exceeding the local record as rewards. Since `epoch_update_rewards()` must be invoked actively and is not triggered automatically, it is important to ensure it is invoked in a timely manner to prevent potential loss of rewards.

2.3.3 Potential arbitrage opportunity

Introduced by `Version 1`

Description The function `epoch_update_rewards()` updates accumulated rewards, which are added to `total_staked_near_amount`, thereby increasing the share price. This could introduce an arbitrage opportunity where a user deposits before the share price update and unstake afterward, potentially earning a risk-free profit.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS