Security Audit Report for Octopus Restaking

Date: Feb 02, 2024

Version: 1.0

Contact: contact@blocksec.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2024.02.04 10:51:27 +08'00'

Contents

Report Manifest

Item	Description
Client	Octopus-Network
Target	Octopus Restaking

Version History

Version	Date	Description
1.0	February 02, 2024	First Version

About BlockSec The BlockSec Team focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and released detailed analysis reports of high-impact security incidents. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Rust
Approach	Semi-automatic and manual verification

The repository that has been audited includes the lpos_market [1] and restaking-base [2].

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

Repository		Commit SHA
lpos_market	`Version 1`	`00c949fe490ca362f94f29d352604549c87b0b79`
	`Version 2`	`76500e0207647c159700e69833ab8c87b0f65221`
restaking-base	`Version 1`	`bd12a2209d919150362f93e347dc5fe7f30878a5`
	`Version 2`	`55eaa5c77ae17db82c9240625c1a2fc00ded795e`

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] https://github.com/octopus-network/lpos_market

[2] https://github.com/octopus-network/restaking-base

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

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

We show the main concrete checkpoints in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Access control
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [3] and Common Weakness Enumeration [4]. 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.

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

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

Chapter 2 Findings

In total, we find **thirty-two** potential issues. Besides, we have **six** recommendations as follows:

- High Risk: 19
- Medium Risk: 9
- Low Risk: 4
- Recommendations: 6
- Notes: 1

ID	Severity	Description	Category	Status
1	High	Incorrect Caller Verification in Function ft_on_transfer()	DeFi Security	Fixed
2	High	Lack of Check in Function slash_request()	DeFi Security	Fixed
3	High	Lack of Check in Function handle_anchor_deposit_reward_msg()	DeFi Security	Confirmed
4	High	Incorrect Calculation of Validator Commission	DeFi Security	Fixed
5	Low	Incorrect Validation in Function deploy()	DeFi Security	Confirmed
6	Medium	Failure of Cross-Contract Call Result Handling	DeFi Security	Confirmed
7	Medium	Incorrect Validation in Function delegate()	DeFi Security	Confirmed
8	High	Incorrect Slash Amount	DeFi Security	Fixed
9	High	Incorrect Use of max()/min()	DeFi Security	Fixed
10	High	Funds Loss Due to Unsaved Treasury Account	DeFi Security	Fixed
11	High	Potential DoS Due to Inappropriate Implementation of Locking Logic	DeFi Security	Confirmed
12	High	Unrefunded NEAR in function stake_after_check_whitelisted()	DeFi Security	Fixed
13	Medium	Incorrect Rounding Direction	DeFi Security	Fixed
14	Medium	Potential Panic in Callback Function bond_callback()	DeFi Security	Fixed
15	High	Ineffective Lock on Important Functions	DeFi Security	Fixed
16	Low	Lack of Pause Functionality in Function bond()	DeFi Security	Fixed
17	Medium	Unrefunded Storage Fee of Failed Cross-contract Invocations	DeFi Security	Confirmed
18	High	Incorrect Amount of NEAR Attached in Function change_key()	DeFi Security	Fixed
19	Medium	Incorrect Gas Setting in the Function bond()	DeFi Security	Fixed
20	Medium	Unintended Overpayment of Fees by Contract Account	DeFi Security	Fixed
21	High	Failure to Clear State Due to Delayed State Saving	DeFi Security	Fixed
22	High	Potential DoS in internal_slash_in_staker_shares()	DeFi Security	Fixed

23	High	Incorrect Penalty Amount in the Slash Process	DeFi Security	Fixed
24	Medium	Unlimited Delay in Asset Withdrawal Due to Continuous Invocation of decease_stake()	DeFi Security	Fixed
25	Low	Unrefunded STORAGE_FEE of Released Storage	DeFi Security	Confirmed
26	Low	No Storage Fee Charged in Function sync_consumer_chain_pos()	DeFi Security	Confirmed
27	High	Unlimited Withdrawn with Reused Unstake-BatchId	DeFi Security	Fixed
28	High	Potential Panic in Callback Function stake_after_check_whitelisted()	DeFi Security	Fixed
29	Medium	Lack of Storage Fee Charge	DeFi Security	Confirmed
30	High	Panic in Callback Function stake_callback()	DeFi Security	Fixed
31	High	Potential DoS in Function destroy()	DeFi Security	Confirmed
32	High	Potential Panic in Function transfer_near()	DeFi Security	Confirmed
33		Redundant code	Recommendation	Fixed
34	-	Lack of Validation for Register Fee	Recommendation	Confirmed
35	-	Lack of Check in Function delegate	Recommendation	Fixed
36	-	Incorrect Error Message	Recommendation	Fixed
37	-	Refunding Excessive Registration Fee to Incorrect Recipients	Recommendation	Fixed
38	-	Lack of Check on Account's NEAR Balance	Recommendation	Fixed
39	-	TPotential Centralization Problem	Note	Confirmed

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Incorrect Caller Verification in Function ft_on_transfer()

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Function `ft_on_transfer()` of contract `LposMarket` is used to handle received reward tokens. However, the `sender_id` is not well checked. Note that the check `sender_id` is equal to `anchor_id` can be bypassed. This is because the `anchor_id` is also retrieved from the variable `msg` and a malicious user can feed any `msg`. In this case, anyone can distribute any tokens to the protocol.

```
 8   fn ft_on_transfer(
 9       &mut self,
10       sender_id: AccountId,
11       amount: U128,
12       msg: String,
13   ) -> PromiseOrValue<U128> {
14       let deposit_reward_msg =
15           near_sdk::serde_json::from_str::<FtTransferMessage>(&msg).expect("Invalid msg");
16
17
18       match deposit_reward_msg {
19           FtTransferMessage::AnchorDepositRewardMsg(msg) => {
20               self.handle_anchor_deposit_reward_msg(sender_id, amount.0, msg)
21           }
22       }
23   }
```

Listing 2.1: near_ft_impl.rs

```
27   fn handle_anchor_deposit_reward_msg(
28       &mut self,
29       sender_id: AccountId,
30       amount: u128,
31       msg: AnchorDepositRewardMsg,
32   ) -> PromiseOrValue<U128> {
33       // check is anchor
34       let consumer_chain_id = msg.consumer_chain_id;
35       let anchor_id = self
36           .consumer_chains
37           .get(&consumer_chain_id)
38           .unwrap()
39           .anchor_id;
40       assert_eq!(sender_id, anchor_id);
41
42
43       self.anchor_deposit_rewards.push(&AnchorDepositRewardInfo {
```

```
            consumer_chain_id: consumer_chain_id,
            anchor_id: sender_id,
            reward_token_id: env::predecessor_account_id(),
            reward_amount: amount,
            validator_set: msg.validator_set,
            sequence: msg.sequence,
        });
        PromiseOrValue::Value(0.into())
    }
```

Listing 2.2: near_ft_impl.rs

```
    fn distribute_latest_reward(&mut self) {
        self.assert_contract_is_running();
        let reward_info = self.anchor_deposit_rewards.pop().expect("No reward found.");
        log!("Reward information: {:?}", reward_info);


        let total_stake_amount: u128 = reward_info.validator_set.iter().map(|(_, b)| b.0).sum();


        assert!(total_stake_amount > 0);


        let octopus_commission_amount = (U256::from(reward_info.reward_amount)
            * U256::from(self.settings.octopus_commission_rate)
            / U256::from(100))
        .as_u128();
        let remain_reward_amount = reward_info
            .reward_amount
            .checked_sub(octopus_commission_amount)
            .unwrap();


        self.internal_deposit_octopus_commission(
            &reward_info.reward_token_id,
            octopus_commission_amount,
        );


        for (escrow_id, stake_amount) in reward_info.validator_set {
            let validator_id = self.escrow_validators.get(&escrow_id).unwrap();
            let validator_receive_reward_amount = (U256::from(stake_amount.0)
                * U256::from(remain_reward_amount)
                / U256::from(total_stake_amount))
            .as_u128();


            self.internal_use_validator_or_panic(&validator_id, |validator| {
                validator.restaking_rewards.push(&RewardInfo {
                    reward_token_id: reward_info.reward_token_id.clone(),
                    reward_amount: validator_receive_reward_amount,
                    sequence: reward_info.sequence.clone(),
```

```
110                 claimed_delegators: UnorderedSet::new(
111                     StorageKey::RewardClaimedDelegatorInValidator {
112                         validator_id: validator.validator_id.clone(),
113                         timestamp: env::block_timestamp(),
114                     },
115                 ),
116             })
117         });
118     }
119 }
```

Listing 2.3: restaking_impl.rs

Impact Malicious users can add a large number of fake `AnchorDepositRewardInfo` entries to `anchor_deposit_rewards`, affecting reward distribution.

Suggestion Implement a whitelist for tokens that serve as rewards.

2.1.2 Lack of Check in Function slash_request()

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `slash_request()` of contract `RestakingBaseContract`, there is no check to ensure that there is a bond relationship between the slash object in the parameter `slash_items` and the `consumer_chain`.

Malicious users are able to become the Gov Account of a `consumer_chain` by utilizing the function `register_consumer_chain()`. Subsequently, by invoking the functions `slash_request()` and `slash()`, they can carry out a slash operation on any staker to steal assets.

```
59     #[payable]
60     fn register_consumer_chain(&mut self, register_param: ConsumerChainRegisterParam) {
61         self.assert_contract_is_running();
62         // check register_fee eq env::attached_deposit
63         assert_eq!(
64             env::attached_deposit(),
65             self.cc_register_fee,
66             "Attached near should be {}",
67             self.cc_register_fee
68         );
69         // check chain id not used
70         assert!(
71             self.consumer_chains
72                 .get(&register_param.consumer_chain_id)
73                 .is_none(),
74             "This ConsumerChainId {} has been registered.",
75             register_param.consumer_chain_id
76         );
77
78
79         validate_chain_id(&register_param.consumer_chain_id);
```

```
80
81
82          let consumer_chain = ConsumerChain::new_from_register_param(
83              register_param.clone(),
84              env::predecessor_account_id(),
85              self.cc_register_fee,
86          );
87
88
89          // needn't check storage, the register fee should able to cover storage.
90          self.consumer_chains
91              .insert(&consumer_chain.consumer_chain_id, &consumer_chain);
92
93
94          Event::RegisterConsumerChain {
95              consumer_chain_info: &consumer_chain.into(),
96              consumer_chain_register_param: &register_param,
97          }
98          .emit();
99      }
```

Listing 2.4: restaking_impl.rs

```
15      #[payable]
16      fn slash_request(
17          &mut self,
18          consumer_chain_id: ConsumerChainId,
19          slash_items: Vec<(AccountId, U128)>,
20          evidence_sha256_hash: String,
21      ) -> SlashId {
22          self.assert_contract_is_running();
23          assert_eq!(
24              env::attached_deposit(),
25              self.slash_guarantee,
26              "The attached near({}) not equal slash guarantee.({})",
27              env::attached_deposit(),
28              self.slash_guarantee
29          );
30          let consumer_chain = self.internal_get_consumer_chain_or_panic(&consumer_chain_id);
31          consumer_chain.assert_cc_pos_account();
32
33
34          let slash_id = U64(self.next_uuid());
35
36
37          Event::RequestSlash {
38              consumer_chain_id: &consumer_chain_id,
39              slash_items: &near_sdk::serde_json::to_string(&slash_items).unwrap(),
40              evidence_sha256_hash: &evidence_sha256_hash,
41          }
42          .emit();
43
44
```

```
45        // needn't check storage, the slash guarantee should able to cover storage.
46        self.slashes.insert(
47            &slash_id,
48            &Slash {
49                consumer_chain_id,
50                slash_items,
51                evidence_sha256_hash,
52                slash_guarantee: self.slash_guarantee.into(),
53            },
54        );
55
56
57        slash_id
58    }
```

Listing 2.5: restaking_impl.rs

Impact The assets in the protocol can be drained.

Suggestion Add the check to ensure that the `slash` object in `slash_items` has a bond relationship with the `consumer_chain`.

2.1.3 Lack of Check in Function handle_anchor_deposit_reward_msg()

Severity High

Status Confirmed

Introduced by `Version 1`

Description Function `handle_anchor_deposit_reward_msg()` of contract `LposMarket` is designed to record the rewards information. Specifically, a struct of type `AnchorDepositRewardInfo` will be constructed based on the user-provided parameter `msg`, and this struct will be pushed into `anchor_deposit_rewards` for further distribution.

The distribution of rewards is implemented within the function `distribute_latest_reward()`. Before the rewards recorded in `anchor_deposit_rewards` are used, they are checked for validity. Since the rewards recorded in `anchor_deposit_rewards` are read one by one using the function `pop()`, if the validation fails for any of the rewards, causing a revert, the subsequent rewards cannot be distributed.

As the reward distribution is entirely based on the custom msg provided by the user. Malicious users can intentionally construct an invalid `AnchorDepositRewardInfo` to block the whole reward distribution process.

```
8     fn ft_on_transfer(
9         &mut self,
10        sender_id: AccountId,
11        amount: U128,
12        msg: String,
13    ) -> PromiseOrValue<U128> {     let deposit_reward_msg =
14            near_sdk::serde_json::from_str::<FtTransferMessage>(&msg).expect("Invalid msg");
15
16
17        match deposit_reward_msg {
18            FtTransferMessage::AnchorDepositRewardMsg(msg) => {
```

```
19                 self.handle_anchor_deposit_reward_msg(sender_id, amount.0, msg)
20             }
21         }
22     }
```

Listing 2.6: near_ft_impl.rs

```
27     fn handle_anchor_deposit_reward_msg(
28         &mut self,
29         sender_id: AccountId,
30         amount: u128,
31         msg: AnchorDepositRewardMsg,
32     ) -> PromiseOrValue<U128> {
33         // check is anchor
34         let consumer_chain_id = msg.consumer_chain_id;
35         let anchor_id = self
36             .consumer_chains
37             .get(&consumer_chain_id)
38             .unwrap()
39             .anchor_id;
40         assert_eq!(sender_id, anchor_id);
41
42
43         self.anchor_deposit_rewards.push(&AnchorDepositRewardInfo {
44             consumer_chain_id: consumer_chain_id,
45             anchor_id: sender_id,
46             reward_token_id: env::predecessor_account_id(),
47             reward_amount: amount,
48             validator_set: msg.validator_set,
49             sequence: msg.sequence,
50         });
51         PromiseOrValue::Value(0.into())
52     }
```

Listing 2.7: near_ft_impl.rs

```
69     fn distribute_latest_reward(&mut self) {
70         self.assert_contract_is_running();
71         let reward_info = self.anchor_deposit_rewards.pop().expect("No reward found.");
72         log!("Reward information: {:?}", reward_info);
73
74
75         let total_stake_amount: u128 = reward_info.validator_set.iter().map(|(_, b)| b.0).sum();
76
77
78         assert!(total_stake_amount > 0);
79
80
81         let octopus_commission_amount = (U256::from(reward_info.reward_amount)
82             * U256::from(self.settings.octopus_commission_rate)
83             / U256::from(100))
84         .as_u128();
85         let remain_reward_amount = reward_info
```

```
86              .reward_amount
87              .checked_sub(octopus_commission_amount)
88              .unwrap();
89
90
91          self.internal_deposit_octopus_commission(
92              &reward_info.reward_token_id,
93              octopus_commission_amount,
94          );
95
96
97          for (escrow_id, stake_amount) in reward_info.validator_set {
98              let validator_id = self.escrow_validators.get(&escrow_id).unwrap();
99              let validator_receive_reward_amount = (U256::from(stake_amount.0)
100                 * U256::from(remain_reward_amount)
101                 / U256::from(total_stake_amount))
102             .as_u128();
103
104
105             self.internal_use_validator_or_panic(&validator_id, |validator| {
106                 validator.restaking_rewards.push(&RewardInfo {
107                     reward_token_id: reward_info.reward_token_id.clone(),
108                     reward_amount: validator_receive_reward_amount,
109                     sequence: reward_info.sequence.clone(),
110                     claimed_delegators: UnorderedSet::new(
111                         StorageKey::RewardClaimedDelegatorInValidator {
112                             validator_id: validator.validator_id.clone(),
113                             timestamp: env::block_timestamp(),
114                         },
115                     ),
116                 })
117             });
118         }
119     }
```

Listing 2.8: restaking_impl.rs

Impact The reward distribution process is vulnerable to potential `DoS` attacks.

Suggestion Add checks in the function `ft_on_transfer()` to ensure the parameter `msg` is valid.

Feedback from the Project The project team will manually address any cases of DoS that occur within the reward distribution process.

2.1.4 Incorrect Calculation of Validator Commission

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `distribute_latest_reward_in_validator()` of contract `LposMarket`, `validator.restaking_rewards` might pop/push `reward_info` multiple times for recording delegator rewards.

However, after each pop of `reward_info`, the `validator` commission is calculated and collected. This process results in a scenario where, if a `reward_info` is popped multiple times, the `validator_commission_amount` gets recalculated, leading to the `validator` collecting more commission fee than intended.

```
117    fn distribute_latest_reward_in_validator(&mut self, validator_id: AccountId) {
118        self.assert_contract_is_running();
119        let mut validator = self.internal_get_validator_or_panic(&validator_id);
120        let mut reward_info = validator.restaking_rewards.pop().expect("No reward found.");
121        log!("Reward information: {:?}", reward_info);
122
123
124        let validator_state = validator
125            .get_latest_validator_state_le_sequence(reward_info.sequence.0)
126            .unwrap();
127
128
129        let validator_commission_amount = (U256::from(reward_info.reward_amount)
130            * U256::from(self.settings.validator_commission_rate)
131            / U256::from(100))
132        .as_u128();
133
134
135        validator.deposit_reward( \
136            &reward_info.reward_token_id,
137            validator_commission_amount,
138            RewardSource::ValidatorShared,
139        );
140
141
142        let remain_reward_amount = reward_info
143            .reward_amount
144            .checked_sub(validator_commission_amount)
145            .unwrap();
146
147
148        let delegator_set_option =
149            validator.get_latest_delegator_set_le_sequence(reward_info.sequence.0);
150
151
152        if let Some(delegator_set) = delegator_set_option {
153            let mut distribute_reward_to_delegator_times = 0;
154            for delegator_id in &delegator_set.delegators {
155                if reward_info.claimed_delegators.contains(delegator_id) {
156                    continue;
157                }
158
159
160                reward_info.claimed_delegators.insert(&delegator_id);
161
162
163                let delegator_shares = validator
164                    .get_latest_delegator_shares_le_sequence(delegator_id, reward_info.sequence.0)
165                    .unwrap();
```

```
166                 let receive_reward_amount = (U256::from(delegator_shares)
167                     * U256::from(remain_reward_amount)
168                     / U256::from(validator_state.total_share_balance))
169                 .as_u128();
170                 self.internal_use_delegator_or_panic(delegator_id, |delegator| {
171                     delegator.deposit_reward(
172                         &reward_info.reward_token_id,
173                         receive_reward_amount,
174                         &validator_id,
175                     );
176                 });
177
178
179                 distribute_reward_to_delegator_times += 1;
180                 if distribute_reward_to_delegator_times ==
                        MAX_TIMES_OF_DISTRIBUTE_DELEGATORS_REWARD
181                 {
182                     break;
183                 }
184             }
185
186
187             if (reward_info.claimed_delegators.len() as usize) < delegator_set.delegators.len() {
188                 validator.restaking_rewards.push(&reward_info);
189                 self.internal_save_validator(&validator);
190                 log!(
191                     "Distribute {} times to delegators",
192                     distribute_reward_to_delegator_times
193                 );
194                 return;
195             }
196         }
197
198
199         let validator_receive_reward_amount = (U256::from(validator_state.validator_share_balance)
200             * U256::from(remain_reward_amount)
201             / U256::from(validator_state.total_share_balance))
202         .as_u128();
203         validator.deposit_reward(
204             &reward_info.reward_token_id,
205             validator_receive_reward_amount,
206             RewardSource::Restaking,
207         );
208
209
210         reward_info.claimed_delegators.clear();
211         self.internal_save_validator(&validator);
212     }
```

Listing 2.9: restaking_impl.sol

Impact `Validators` charge more commissions than expected.

Suggestion Revise the calculation logic for `Validator Commission`.

2.1.5 Incorrect Validation in Function deploy()

Severity Low

Status Confirmed

Introduced by `Version 1`

Description Validators can deploy `escrow` contracts for themselves by invoking the function `deploy()` of the contract `LposMarket`, and a validator can only deploy one `escrow` contract. This is ensured by checking whether the `validator` already exists in the `self.validator` (lines 22 -24).

However, the update of `self.validators` occurs in the callback function of function `deploy()`. Users can bypass this check by invoking the function `deploy()` multiple times within a single block, allowing them to deploy multiple `escrow` contracts.

```
15    #[payable]
16    fn deploy(&mut self) {
17        self.assert_contract_is_running();
18        assert_eq!(env::attached_deposit(), self.settings.validator_deploy_fee);
19
20
21        let account_id = env::predecessor_account_id();
22        if self.validators.get(&account_id).is_some() {
23            return;
24        }
25
26
27        let escrow_id = AccountId::new_unchecked(
28            format!("{}.{}", self.next_uuid(), env::current_account_id()).to_string(),
29        );
30        Promise::new(escrow_id.clone())
31            .create_account()
32            .transfer(self.deploy_escrow_amount())
33            .deploy_contract(
34                env::storage_read(&borsh::to_vec(&StorageKey::EscrowContractWasm).unwrap())
35                    .unwrap(),
36            )
37            .function_call(
38                "init".to_string(),
39                json!({
40                    "validator": env::predecessor_account_id(),
41                    "lpos_market_contract": env::current_account_id(),
42                    "restaking_base_contract": self.settings.restaking_base_contract.clone()
43                })
44                .to_string()
45                .into_bytes(),
46                0,
47                Gas::ONE_TERA.mul(TGAS_FOR_INIT_ESCROW_CONTRACT),
48            )
49            .then(
50                Self::ext(env::current_account_id())
51                    .deploy_callback(env::predecessor_account_id(), escrow_id.clone()),
52            )
53            .then(
```

```
54              ext_restaking_base::ext(self.settings.restaking_base_contract.clone())
55                  .with_attached_deposit(RESTAKING_BASE_REGISTER_FEE)
56                  .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_STORAGE_DEPOSIT))
57                  .storage_deposit(Some(escrow_id), None),
58          );
59  }
```

Listing 2.10: staking_impl.rs

Impact A validator can deploy multiple `escrow` contracts.

Suggestion Move the check into the callback function.

Feedback from the Project The team thinks this issue will not affect the actual process.

2.1.6 Failure of Cross-Contract Call Result Handling

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description In the function `deploy()` of the contract `LposMarket`, the `escrow` contract is first deployed via a cross-contract call using the function `deploy_contract()`, followed by another cross-contract invocation of the function `storage_deposit()` of the contract `restaking_base` with storage fee attached.

However, there is a lack of verification for the promise result returned by the function `deploy_contract()` before the function `storage_deposit()` is invoked. Therefore, if the deployment of the `escrow` contract fails, the storage fee would still be deposited into the contract `restaking_base`.

```
15  #[payable]
16  fn deploy(&mut self) {
17      self.assert_contract_is_running();
18      assert_eq!(env::attached_deposit(), self.settings.validator_deploy_fee);
19
20
21      let account_id = env::predecessor_account_id();
22      if self.validators.get(&account_id).is_some() {
23          return;
24      }
25
26
27      let escrow_id = AccountId::new_unchecked(
28          format!("{}.{}", self.next_uuid(), env::current_account_id()).to_string(),
29      );
30      Promise::new(escrow_id.clone())
31          .create_account()
32          .transfer(self.deploy_escrow_amount())
33          .deploy_contract(
34              env::storage_read(&borsh::to_vec(&StorageKey::EscrowContractWasm).unwrap())
35                  .unwrap(),
36          )
37          .function_call(
38              "init".to_string(),
39              json!({
```

```
40                  "validator": env::predecessor_account_id(),
41                  "lpos_market_contract": env::current_account_id(),
42                  "restaking_base_contract": self.settings.restaking_base_contract.clone()
43              })
44              .to_string()
45              .into_bytes(),
46              0,
47              Gas::ONE_TERA.mul(TGAS_FOR_INIT_ESCROW_CONTRACT),
48          )
49          .then(
50              Self::ext(env::current_account_id())
51                  .deploy_callback(env::predecessor_account_id(), escrow_id.clone()),
52          )
53          .then(
54              ext_restaking_base::ext(self.settings.restaking_base_contract.clone())
55                  .with_attached_deposit(RESTAKING_BASE_REGISTER_FEE)
56                  .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_STORAGE_DEPOSIT))
57                  .storage_deposit(Some(escrow_id), None),
58          );
59  }
```

Listing 2.11: staking_impl.rs

Impact The validator will lose the storage fee if the deployment of the `escrow` contract fails.

Suggestion Implement corresponding logic to handle the failed promise result.

2.1.7 Incorrect Validation in Function delegate()

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description In the function `delegate()` of the contract `LposMarket`, a check is performed on the number of `delegators` for a `validator`, ensuring that it does exceed the specified `maximum_delegators_limit` (lines 270 - 275). However, the check is implemented incorrectly.

First, it allows new `delegators` to delegate to a `validator` even when the number of `delegators` is equal to the `maximum_delegators_limit`. Second, since the length of the `delegators` associated with a `validator` is updated in the callback function `delegate_callback()`, if different `delegators` invoke this function within the same block, the aforementioned check can still be bypassed, even if the `maximum limit` for `delegators` has been reached.

```
253  #[payable]
254  fn delegate(&mut self, validator_id: AccountId, delegate_amount: U128) {
255      self.assert_contract_is_running();
256      self.assert_max_gas();
257
258
259      assert!(delegate_amount.0 > 0);
260      assert_eq!(STORAGE_FEE + delegate_amount.0, env::attached_deposit());
261
262
```

```
263         let delegator_id = env::predecessor_account_id();
264         let delegator = self.internal_get_delegator_or_insert_new(&delegator_id);
265
266
267         assert!(
268             delegator.share_balance == 0 && delegator.select_validator_id.is_none(),
269             "Failed to delegate, already stake in {:?}.",
270             delegator.select_validator_id
271         );
272
273
274         assert!(
275             delegate_amount.0 >= self.settings.minimum_delegator_stake_amount,
276             "Failed to stake, attach near({}) less than minimum_delegator_stake_amount.({}) ",
277             env::attached_deposit(),
278             self.settings.minimum_delegator_stake_amount
279         );
280         let validator = self.internal_get_validator_or_panic(&validator_id);
281
282
283         assert!((validator.delegators.len() as u32) <= self.settings.maximum_delegators_limit);
284         assert!(matches!(validator.status, ValidatorStatus::Staking));
285
286
287         self.ping(validator.validator_id).then(
288             Self::ext(env::current_account_id())
289                 .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_INCREASE_STAKE_AFTER_PING))
290                 .delegate_after_ping(delegator_id, validator_id, delegate_amount),
291         );
292     }
```

Listing 2.12: staking_impl.rs

```
419     #[private]
420     fn delegate_callback(
421         &mut self,
422         delegator_id: AccountId,
423         validator_id: AccountId,
424         delegation_amount: U128,
425     ) {
426         match env::promise_result(0) {
427             PromiseResult::NotReady => unreachable!(),
428             PromiseResult::Successful(value) => {
429                 if let Some(staking_change_result) =
430                     from_slice::<Option<StakingChangeResult>>(&value).unwrap()
431                 {
432                     let mut delegator = self.internal_get_delegator_or_panic(&delegator_id);
433
434
435                     self.internal_use_validator_or_panic(&validator_id, |validator| {
436                         let increase_shares = validator.delegate(
437                             &mut delegator,
438                             delegation_amount.0,
```

```
439                         staking_change_result.sequence.0,
440                         staking_change_result.new_total_staked_balance.0,
441                     );
442                     Event::Delegate {
443                         validator_info: &validator.into(),
444                         delegator_info: &(&delegator).into(),
445                         delegate_amount: &delegation_amount,
446                         increase_shares: &increase_shares.into(),
447                     }
448                     .emit();
449                 });
450
451
452                 self.internal_save_delegator(&delegator);
453             } else {
454                 // the near have refund in escrow contract
455                 emit_callback_failed_event();
456             }
457         }
458         PromiseResult::Failed => {
459             // the near have refund in escrow contract
460             emit_callback_failed_event();
461         }
462     }
463 }
```

Listing 2.13: staking_impl.rs

Impact The length of `validator.delegators` can exceed the `maximum_delegators_limit`.

Suggestion The check should be implemented in the callback function `delegate_callback()`, and it should only be performed when the delegation is successful.

Feedback from the Project The team plans to remove `maximum_delegators_limit` in the future.

2.1.8 Incorrect Slash Amount

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `RestakingBaseContract`, the function `internal_slash()` is used to penalize malicious `validators`. Based on the input `slash_amount`, it first deducts from the validator's `pending_withdrawals`, then from the `validator`'s shares, until the deducted amount equals to `slash_amount`. However, the variable `slashed_amount_from_pending_withdrawals`, which records the amount that has been slashed from `pending_withdrawals`, is not counted in the process of slashing user shares. Therefore, the assets that are actually slashed are incorrect.

```
365     pub(crate) fn internal_slash(
366         &mut self,
367         slash_staker_id: &StakerId,
368         slash_amount: Balance,
```

```
369         treasury: &AccountId,
370     ) -> Balance {
371         let staker = self.internal_get_staker_or_panic(slash_staker_id);
372
373
374         // 1. staker pending withdrawals
375         let slashed_amount_from_pending_withdrawals =
376             self.internal_slash_in_pending_withdrawals(slash_staker_id, slash_amount, treasury);
377
378
379         if slashed_amount_from_pending_withdrawals == slash_amount {
380             return slash_amount;
381         }
382
383
384         let slashed_amount_from_staker_shares = if staker.shares != 0 {
385             self.internal_slash_in_staker_shares(slash_staker_id, slash_amount, treasury)
386         } else {
387             0
388         };
389         return slashed_amount_from_pending_withdrawals + slashed_amount_from_staker_shares;
390     }
```

Listing 2.14: restaking_impl.rs

Impact The amount by which users are slashed is higher than expected.

Suggestion When slashing from user shares, the amount slashed from the `pending_withdrawals` should be included in the calculation.

2.1.9 Incorrect Use of max()/min()

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In `restaking_impl.rs`, the function `internal_slash_in_pending_withdrawals()` is used to slash a validator's `pending_withdrawals`.`acc_slash_amount` represents the `accumulated_slash_amount`, and the difference between `slash_amount` and `acc_slash_amount` represents the remaining amount that needs to be slashed.

Therefore, when slashing each `pending_withdrawal`, the amount to be slashed should be the minimum value between the quantity of assets within that `pending_withdrawal` and the remaining amount that needs to be slashed. This ensures that the slashing process will not be reverted, even if the quantity of assets within the `pending_withdrawal` is insufficient or only a portion of the `pending_withdrawal's` assets needs to be slashed. However, the current implementation uses function `max()` to take the larger number between them when slashing.

```
389     pub(crate) fn internal_slash_in_pending_withdrawals(
390         &mut self,
391         slash_staker_id: &StakerId,
392         slash_amount: Balance,
```

```
393         treasury: &AccountId,
394     ) -> Balance {
395         let staker_account = self.internal_get_account_or_panic(slash_staker_id);
396         let mut treasury_account = self.internal_get_account_or_new(&treasury);
397         let mut pending_withdrawals = staker_account
398             .pending_withdrawals
399             .values()
400             .sorted_by(|a, b| a.unlock_time.cmp(&b.unlock_time))
401             .collect_vec();
402
403
404         let mut acc_slash_amount = 0;
405         for pending_withdrawal in &mut pending_withdrawals {
406             if acc_slash_amount == slash_amount {
407                 break;
408             }
409             let new_pending_withdrawal = pending_withdrawal.slash(
410                 self.next_uuid().into(),
411                 max(pending_withdrawal.amount, slash_amount - acc_slash_amount),
412                 treasury.clone(),
413             );
414
415
416             treasury_account.pending_withdrawals.insert(
417                 &new_pending_withdrawal.withdrawal_certificate,
418                 &new_pending_withdrawal,
419             );
420             acc_slash_amount += new_pending_withdrawal.amount;
421         }
422         acc_slash_amount
423     }
```

Listing 2.15: restaking_impl.rs

```
46     pub fn slash(
47         &mut self,
48         withdrawal_certificate: WithdrawalCertificate,
49         amount: Balance,
50         beneficiary: AccountId,
51     ) -> Self {
52         self.amount = self.amount
53         .checked_sub(amount)
54         .expect(format!("Failed to slash, the slash amount({}) is greater than PendingWithdrawal
               amount({})", amount, self.amount)
55         .as_str());
56
57
58         Self {
59             withdrawal_certificate,
60             pool_id: self.pool_id.clone(),
61             amount: amount,
62             unlock_epoch: self.unlock_epoch,
63             unlock_time: env::block_timestamp(),
```

```
64              beneficiary,
65              allow_other_withdraw: true,
66          }
67      }
```

Listing 2.16: pending_withdrawal.rs

Impact Using the function `max()` could potentially trigger a panic, preventing the slashing process from proceeding.

Suggestion Replace the function `max()` with `min()`.

2.1.10 Funds Loss Due to Unsaved Treasury Account

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `RestakingBaseContract`, the function `internal_slash_in_pending_withdrawals()` aims to slash malicious validators from their `pending_withdrawals`. The slashed assets will be recorded into the `pending_withdrawals` of `treasury_account` specified by the `consumer_chain's Gov` for further withdrawal. However, the updated account actually is not saved, leading to the loss of the slashed assets.

```
389     pub(crate) fn internal_slash_in_pending_withdrawals(
390         &mut self,
391         slash_staker_id: &StakerId,
392         slash_amount: Balance,
393         treasury: &AccountId,
394     ) -> Balance {
395         let staker_account = self.internal_get_account_or_panic(slash_staker_id);
396         let mut treasury_account = self.internal_get_account_or_new(&treasury);
397         let mut pending_withdrawals = staker_account
398             .pending_withdrawals
399             .values()
400             .sorted_by(|a, b| a.unlock_time.cmp(&b.unlock_time))
401             .collect_vec();
402
403
404         let mut acc_slash_amount = 0;
405         for pending_withdrawal in &mut pending_withdrawals {
406             if acc_slash_amount == slash_amount {
407                 break;
408             }
409             let new_pending_withdrawal = pending_withdrawal.slash(
410                 self.next_uuid().into(),
411                 max(pending_withdrawal.amount, slash_amount - acc_slash_amount),
412                 treasury.clone(),
413             );
414
415
416             treasury_account.pending_withdrawals.insert(
417                 &new_pending_withdrawal.withdrawal_certificate,
```

```
418                 &new_pending_withdrawal,
419             );
420             acc_slash_amount += new_pending_withdrawal.amount;
421         }
422         acc_slash_amount
423     }
```

Listing 2.17: restaking_impl.rs

Impact Slashed assets are lost.

Suggestion Invoke the function `internal_save_account()` to save the updated `treasury_account` after each slashing.

2.1.11 Potential DoS Due to Inappropriate Implementation of Locking Logic

Severity High

Status Confirmed

Introduced by `Version 1`

Description Any user can first invoke the function `storage_deposit()` in `storage_management_impl.rs` to register an `account` for further operations in protocol. Some operations, such as staking, have implemented locking logic to prevent data inconsistency by asynchronous invocations. Specifically, in the process of staking, one of the callback functions `stake_after_ping()` will lock the `staking_pool` before performing the stake operation within that `staking_pool`. When the `staking_pool` is locked, it means that other users cannot perform any other operations on the `staking_pool` until it's unlocked.

In this case, since any user can invoke the function `stake()` after invoking function `storage_deposit()`, and there is no limit on the `stake_amount`, malicious users can repeatedly stake very small amounts of `NEAR`, forcing the `staking_pool's` state continuously locked, which results in a `DoS` attack.

The above issue also occurs in the functions `decrease_stake()`, `increase_stake()`, `increase_delegation()`, and `decrease_delegation()`.

```
11  fn storage_deposit(
12      &mut self,
13      account_id: Option<AccountId>,
14      #[allow(unused)] registration_only: Option<bool>,
15  ) -> StorageBalance {
16      let account_id = account_id.unwrap_or(env::predecessor_account_id());
17      let exist = self.accounts.contains_key(&account_id);
18      if exist {
19          self.transfer_near(account_id.clone(), env::attached_deposit())
20      } else {
21          assert!(env::attached_deposit() >= REGISTER_STORAGE_FEE);
22          self.internal_save_account(&account_id, &Account::new(account_id.clone()));
23          if env::attached_deposit() > REGISTER_STORAGE_FEE {
24              self.transfer_near(
25                  account_id.clone(),
26                  env::attached_deposit() - REGISTER_STORAGE_FEE,
27              )
28          }
29      }
```

```
30
31
32      self.storage_balance_of(account_id).unwrap()
33  }
```

Listing 2.18: storage_management_impl.rs

```
6   fn stake(&mut self, pool_id: PoolId) -> PromiseOrValue<Option<StakingChangeResult>> {
7       self.assert_contract_is_running();
8       assert_attached_near();
9
10
11      let staker_id = env::predecessor_account_id();
12
13
14      assert!(
15          self.accounts.get(&staker_id).is_some(),
16          "Should register by storage_deposit first."
17      );
18
19
20      let staker = self
21          .stakers
22          .get(&staker_id)
23          .unwrap_or(Staker::new(staker_id.clone()));
24
25
26      assert_eq!(staker.shares, 0, "Can't stake, shares is not zero");
27      assert!(
28          staker.select_staking_pool.is_none()
29              || staker.select_staking_pool.clone().unwrap().ne(&pool_id),
30          "Staker({}) have selected pool({})",
31          staker_id,
32          pool_id
33      );
34
35
36      self.internal_save_staker(&staker_id, &staker);
37
38
39      return ext_whitelist::ext(self.staking_pool_whitelist_account.clone())
40          .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_IS_WHITELISTED))
41          .is_whitelisted(pool_id.clone())
42          .then(
43              Self::ext(env::current_account_id())
44                  .with_attached_deposit(env::attached_deposit())
45                  .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_SELECT_POOL_AFTER_CHECK_WHITELIST))
46                  .stake_after_check_whitelisted(staker_id.clone(), pool_id.clone()),
47          )
48          .into();
49  }
```

Listing 2.19: staking_impl.rs

```
538     fn stake_after_ping(
539         &mut self,
540         staker_id: AccountId,
541     ) -> PromiseOrValue<Option<StakingChangeResult>> {
542         match env::promise_result(0) {
543             PromiseResult::NotReady => unreachable!(),
544             PromiseResult::Successful(_) => {
545                 let pool_id =
546                     self.internal_use_staker_staking_pool_or_panic(&staker_id, |staking_pool| {
547                         staking_pool.lock();
548                         staking_pool.pool_id.clone()
549                     });
550
551
552                 ext_staking_pool::ext(pool_id)
553                     .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_DEPOSIT_AND_STAKE))
554                     .with_attached_deposit(env::attached_deposit())
555                     .deposit_and_stake()
556                     .function_call(
557                         "get_account_staked_balance".to_string(),
558                         json!({ "account_id": env::current_account_id() })
559                             .to_string()
560                             .into_bytes(),
561                         0,
562                         Gas::ONE_TERA.mul(TGAS_FOR_GET_ACCOUNT_STAKED_BALANCE),
563                     )
564                     .then(
565                         Self::ext(env::current_account_id())
566                             .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_INCREASE_STAKE_CALL_BACK))
567                             .stake_callback(staker_id, env::attached_deposit().into()),
568                     )
569                     .into()
570             }
571             PromiseResult::Failed => {
572                 log!("Failed to increase stake by ping error.");
573                 self.transfer_near(staker_id, env::attached_deposit());
574                 emit_callback_failed_event();
575                 return PromiseOrValue::Value(None);
576             }
577         }
578     }
```

Listing 2.20: staking_impl.rs

```
538     fn stake_after_ping(
539         &mut self,
540         staker_id: AccountId,
541     ) -> PromiseOrValue<Option<StakingChangeResult>> {
542         match env::promise_result(0) {
543             PromiseResult::NotReady => unreachable!(),
544             PromiseResult::Successful(_) => {
545                 let pool_id =
546                     self.internal_use_staker_staking_pool_or_panic(&staker_id, |staking_pool| {
```

```
                    staking_pool.lock();
                    staking_pool.pool_id.clone()
                });


            ext_staking_pool::ext(pool_id)
                .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_DEPOSIT_AND_STAKE))
                .with_attached_deposit(env::attached_deposit())
                .deposit_and_stake()
                .function_call(
                    "get_account_staked_balance".to_string(),
                    json!({ "account_id": env::current_account_id() })
                        .to_string()
                        .into_bytes(),
                    0,
                    Gas::ONE_TERA.mul(TGAS_FOR_GET_ACCOUNT_STAKED_BALANCE),
                )
                .then(
                    Self::ext(env::current_account_id())
                        .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_INCREASE_STAKE_CALL_BACK))
                        .stake_callback(staker_id, env::attached_deposit().into()),
                )
                .into()
        }
        PromiseResult::Failed => {
            log!("Failed to increase stake by ping error.");
            self.transfer_near(staker_id, env::attached_deposit());
            emit_callback_failed_event();
            return PromiseOrValue::Value(None);
        }
    }
}
```

Listing 2.21: staking_impl.rs

```
 fn decrease_stake(
    &mut self,
    decrease_amount: U128,
    beneficiary: Option<AccountId>,
) -> PromiseOrValue<Option<StakingChangeResult>> {
    self.assert_contract_is_running();
    self.assert_attached_storage_fee();
    assert!(decrease_amount.0 > 0, "The decrease amount should gt 0");


    let staker_id = env::predecessor_account_id();


    self.internal_use_staker_staking_pool_or_panic(&staker_id, |staking_pool| {
        staking_pool.lock()
    });


```

```
114         return self
115             .ping(Option::None)
116             .then(
117                 Self::ext(env::current_account_id())
118                     .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_DECREASE_STAKE_AFTER_PING))
119                     .decrease_stake_after_ping(
120                         staker_id,
121                         decrease_amount,
122                         beneficiary.unwrap_or(env::predecessor_account_id()),
123                     ),
124             )
125             .into();
126     }
```

Listing 2.22: staking_impl.rs

```
75      fn increase_stake(&mut self) -> PromiseOrValue<Option<StakingChangeResult>> {
76          self.assert_contract_is_running();
77          assert_attached_near();
78
79
80          let staker_id = env::predecessor_account_id();
81          self.internal_use_staker_staking_pool_or_panic(&staker_id, |staking_pool| {
82              staking_pool.lock()
83          });
84
85
86          return self
87              .ping(Option::None)
88              .then(
89                  Self::ext(env::current_account_id())
90                      .with_attached_deposit(env::attached_deposit())
91                      .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_INCREASE_STAKE_AFTER_PING))
92                      .increase_stake_after_ping(staker_id),
93              )
94              .into();
95      }
```

Listing 2.23: staking_impl.rs

```
289     fn increase_delegation(&mut self, increase_amount: U128) {
290         self.assert_contract_is_running();
291         self.assert_max_gas();
292         assert!(increase_amount.0 > 0);
293         assert_eq!(STORAGE_FEE + increase_amount.0, env::attached_deposit());
294
295
296         let delegator_id = env::predecessor_account_id();
297         let delegator = self.internal_get_delegator_or_panic(&delegator_id);
298         let validator =
299             self.internal_get_validator_or_panic(&delegator.select_validator_id.unwrap());
300
301
```

```
302     assert!(matches!(validator.status, ValidatorStatus::Staking));
303
304
305     self.ping(validator.validator_id).then(
306         Self::ext(env::current_account_id())
307             .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_INCREASE_STAKE_AFTER_PING))
308             .increase_delegation_after_ping(delegator_id, increase_amount),
309     );
310 }
```

Listing 2.24: staking_impl.rs

```
596  fn increase_delegation_after_ping(&mut self, delegator_id: AccountId, increase_amount: U128) {
597     match env::promise_result(0) {
598         PromiseResult::NotReady => unreachable!(),
599         PromiseResult::Successful(_) => {
600             let (_, validator) =
601                 self.internal_get_delegator_and_validator_or_panic(&delegator_id);
602
603
604             ext_escrow::ext(validator.escrow_id)
605                 .with_attached_deposit(increase_amount.0)
606                 .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_INCREASE_STAKE))
607                 .with_unused_gas_weight(0)
608                 .increase_stake_with_sender(delegator_id.clone())
609                 .then(
610                     Self::ext(env::current_account_id())
611                         .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_INCREASE_STAKE_CALL_BACK))
612                         .increase_delegation_callback(delegator_id, increase_amount),
613                 );
614         }
615         PromiseResult::Failed => {
616             emit_callback_failed_event();
617             self.transfer_near(delegator_id, increase_amount.0);
618         }
619     }
620 }
```

Listing 2.25: staking_impl.rs

```
73  pub fn increase_stake_with_sender(
74     &mut self,
75     sender: AccountId,
76 ) -> PromiseOrValue<Option<StakingChangeResult>> {
77     self.assert_lpos_market();
78     assert_attached_near();
79     log!("Attach gas: {:?}", env::prepaid_gas());
80     return ext_escrow::ext(self.restaking_base_contract.clone())
81         .with_attached_deposit(env::attached_deposit())
82         .with_static_gas(Gas::ONE_TERA.mul(110))
83         .increase_stake()
84         .then(
85             Self::ext(env::current_account_id())
```

```
86                  .with_unused_gas_weight(0)
87                  .with_static_gas(Gas::ONE_TERA.mul(7))
88                  .stake_callback(env::attached_deposit().into(), sender),
89          )
90          .into();
91  }
```

Listing 2.26: lib.rs

```
310   fn decrease_delegation(&mut self, decrease_amount: U128) {
311      self.assert_contract_is_running();
312      self.assert_max_gas();
313      assert_eq!(env::attached_deposit(), STORAGE_FEE);
314      let delegator_id = env::predecessor_account_id();
315      let delegator = self.internal_get_delegator_or_panic(&delegator_id);
316      let validator =
317          self.internal_get_validator_or_panic(&delegator.select_validator_id.unwrap());
318      assert!(matches!(validator.status, ValidatorStatus::Staking));
319      self.ping(validator.validator_id).then(
320          Self::ext(env::current_account_id())
321              .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_DECREASE_STAKE_AFTER_PING))
322              .decrease_delegation_after_ping(delegator_id, decrease_amount),
323      );
324  }
```

Listing 2.27: staking_impl.rs

```
720   fn decrease_delegation_after_ping(&mut self, delegator_id: AccountId, decrease_amount: U128) {
721      match env::promise_result(0) {
722          PromiseResult::NotReady => unreachable!(),
723          PromiseResult::Successful(_) => {
724              let (mut delegator, validator) =
725                  self.internal_get_delegator_and_validator_or_panic(&delegator_id);
726
727
728              let decrease_shares: u128 = validator.calculate_decrease_shares(decrease_amount.0);
729
730
731              let is_able_decrease = delegator.share_balance >= decrease_shares
732                  && validator.calculate_near_balance(delegator.share_balance - decrease_shares)
733                      >= self.settings.minimum_delegator_stake_amount;
734
735
736              if is_able_decrease {
737                  delegator.share_balance -= decrease_shares;
738                  self.internal_save_delegator(&delegator);
739
740
741                  ext_escrow::ext(validator.escrow_id)
742                      .with_attached_deposit(RESTAKING_BASE_STORAGE_FEE)
743                      .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_DECREASE_STAKE))
744                      .with_unused_gas_weight(0)
745                      .decrease_stake(decrease_amount.into(), Some(delegator_id.clone()))
```

```
746                     .then(
747                         Self::ext(env::current_account_id())
748                             .with_static_gas(
749                                 Gas::ONE_TERA.mul(TGAS_FOR_DECREASE_STAKE_CALL_BACK),
750                             )
751                             .decrease_delegation_callback(
752                                 delegator_id.clone(),
753                                 validator.validator_id,
754                                 decrease_shares.into(),
755                                 decrease_amount,
756                             ),
757                     );
758             } else {
759                 emit_callback_failed_event();
760             }
761         }
762         PromiseResult::Failed => {
763             emit_callback_failed_event();
764         }
765     }
766 }
```

Listing 2.28: staking_impl.rs

```
63  fn decrease_stake(
64      &mut self,
65      decrease_amount: U128,
66      beneficiary: Option<AccountId>,
67  ) -> PromiseOrValue<Option<StakingChangeResult>> {
68      self.assert_lpos_market();
69      assert_attached_storage_fee();
70      return ext_escrow::ext(self.restaking_base_contract.clone())
71          .with_attached_deposit(STORAGE_FEE)
72          .decrease_stake(decrease_amount, beneficiary)
73          .into();
74  }
```

Listing 2.29: interface.rs

Impact Malicious users can continuously keep the staking pool in a locked state by repeatedly staking very small amounts of NEAR (at minimal cost), preventing other users from accessing it.

Suggestion Introduce a minimum value check to ensure that the above function is not misused.

Feedback from the Project The team is aware of the potential impact this issue may cause, and have thoroughly discussed the potential additional effects of a minimum value for increase/decrease stake.

2.1.12 Unrefunded NEAR in function stake_after_check_whitelisted()

Severity High

Status Fixed in Version 2

Introduced by Version 1

Description In contract `RestakingBaseContract`, the function `stake_after_check_whitelisted()` will proceed with the remaining logic for the stake operation based on the returned promise result of the function `is_whitelisted()`. Since function `stake_after_check_whitelisted()` requires to attach a number of `NEAR` for staking, when function `is_whitelisted()` returns false, the attached `NEAR` should be refunded to the staker. However, the refund logic is not implemented when handling the failed promise result.

```
6   fn stake(&mut self, pool_id: PoolId) -> PromiseOrValue<Option<StakingChangeResult>> {
7       self.assert_contract_is_running();
8       assert_attached_near();
9
10
11      let staker_id = env::predecessor_account_id();
12
13
14      assert!(
15          self.accounts.get(&staker_id).is_some(),
16          "Should register by storage_deposit first."
17      );
18
19
20      let staker = self
21          .stakers
22          .get(&staker_id)
23          .unwrap_or(Staker::new(staker_id.clone()));
24
25
26      assert_eq!(staker.shares, 0, "Can't stake, shares is not zero");
27      assert!(
28          staker.select_staking_pool.is_none()
29              || staker.select_staking_pool.clone().unwrap().ne(&pool_id),
30          "Staker({}) have selected pool({})",
31          staker_id,
32          pool_id
33      );
34
35
36      self.internal_save_staker(&staker_id, &staker);
37
38
39      return ext_whitelist::ext(self.staking_pool_whitelist_account.clone())
40          .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_IS_WHITELISTED))
41          .is_whitelisted(pool_id.clone())
42          .then(
43              Self::ext(env::current_account_id())
44                  .with_attached_deposit(env::attached_deposit())
45                  .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_SELECT_POOL_AFTER_CHECK_WHITELIST))
46                  .stake_after_check_whitelisted(staker_id.clone(), pool_id.clone()),
47          )
48          .into();
49  }
```

Listing 2.30: staking_impl.rs

```
741     fn stake_after_check_whitelisted(
742         &mut self,
743         staker_id: AccountId,
744         pool_id: PoolId,
745         #[callback] whitelisted: bool,
746     ) -> PromiseOrValue<Option<StakingChangeResult>> {
747         if !whitelisted {
748             log!("Failed to select pool, {} is not whitelisted.", pool_id);
749             return PromiseOrValue::Value(None);
750         }
751
752
753         if !self.staking_pools.get(&pool_id).is_some() {
754             self.internal_save_staking_pool(&StakingPool::new(pool_id.clone(), staker_id.clone()));
755             Event::SaveStakingPool { pool_id: &pool_id }.emit();
756         }
757
758
759         self.internal_use_staker_or_panic(&staker_id, |staker| {
760             staker.select_staking_pool = Some(pool_id.clone());
761         });
762
763
764         self.ping(Some(pool_id))
765             .then(
766                 Self::ext(env::current_account_id())
767                     .with_attached_deposit(env::attached_deposit())
768                     .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_INCREASE_STAKE_AFTER_PING))
769                     .stake_after_ping(staker_id),
770             )
771             .into()
772
773
774         // return PromiseOrValue::Value(true);
775     }
```

Listing 2.31: staking_impl.rs

Impact If the pool selected by the user for staking does not pass the `is_whitelisted()` check, the attached `NEAR` will be lost.

Suggestion Refund the `NEAR` to the user if `is_whitelisted()` returns false.

2.1.13 Incorrect Rounding Direction

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `RestakingBaseContract`, the function `unstake()` is used to unstake the assets from the user's selected staking pool. However, in the callback function `unstake_afer_ping()`, there is an

incorrect rounding direction when calculating the amount for unstaking based on the desired shares to be withdrawn. It utilizes the function `staked_amount_from_shares_balance_rounded_up()` which rounds up to obtain the `receive_amount`, leading to an inflated value (line 300).

For the protocol, if the amounts withdrawn by users are consistently larger, it will result in the last user being unable to withdraw their entire balance. Therefore, during the decrease in stake or unstake operations, it is crucial for the contract to prioritize burning more shares and withdrawing a smaller amount of assets.

The above issue also occurs in the function `decrease_stake_after_ping()`.

```
126   fn unstake(
127       &mut self,
128       beneficiary: Option<AccountId>,
129       withdraw_by_anyone: Option<bool>,
130   ) -> PromiseOrValue<Option<StakingChangeResult>> {
131       self.assert_contract_is_running();
132       self.assert_attached_storage_fee();
133       log!("Prepaid gas: {:?}", env::prepaid_gas());
134       let staker_id = env::predecessor_account_id();
135
136
137       self.internal_use_staker_staking_pool_or_panic(&staker_id, |staking_pool| {
138           staking_pool.lock()      //lock pool
139       });
140
141
142       return self
143           .ping(Option::None)
144           .then(
145               Self::ext(env::current_account_id())
146                   .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_UNSTAKE_AFTER_PING))
147                   .unstake_after_ping(
148                       staker_id.clone(),
149                       beneficiary.unwrap_or(staker_id.clone()),
150                       withdraw_by_anyone.unwrap_or(true),
151                   ),
152           )
153           .into();
154   }
```

Listing 2.32: staking_impl.rs

```
282   fn unstake_after_ping(
283       &mut self,
284       staker_id: AccountId,
285       beneficiary: AccountId,
286       withdraw_by_anyone: bool,
287   ) -> PromiseOrValue<Option<StakingChangeResult>> {
288       match env::promise_result(0) {
289           PromiseResult::NotReady => unreachable!(),
290           PromiseResult::Failed => {
291               emit_callback_failed_event();
292               PromiseOrValue::Value(None)
```

```
293             }
294             PromiseResult::Successful(_) => {
295                 let mut staker = self.internal_get_staker_or_panic(&staker_id);
296                 let staking_pool = self.internal_get_staking_pool_by_staker_or_panic(&staker_id);
297
298
299                 let decrease_shares = staker.shares;
300                 let receive_amount =
301                     staking_pool.staked_amount_from_shares_balance_rounded_up(decrease_shares);
302                 staker.shares = 0;
303                 self.internal_save_staker(&staker_id, &staker);
304
305
306                 ext_staking_pool::ext(staking_pool.pool_id.clone())
307                     .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_UNSTAKE))
308                     .unstake(receive_amount.into())
309                     .function_call(
310                         "get_account_staked_balance".to_string(),
311                         json!({ "account_id": env::current_account_id() })
312                             .to_string()
313                             .into_bytes(),
314                         0,
315                         Gas::ONE_TERA.mul(TGAS_FOR_GET_ACCOUNT_STAKED_BALANCE),
316                     )
317                     .then(
318                         Self::ext(env::current_account_id())
319                             .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_UNSTAKE_CALL_BACK))
320                             .unstake_callback(
321                                 staker_id,
322                                 decrease_shares.into(),
323                                 receive_amount.into(),
324                                 beneficiary,
325                                 withdraw_by_anyone,
326                             ),
327                     )
328                     .into()
329             }
330         }
331     }
```

Listing 2.33: staking_impl.rs

```
322     fn decrease_stake_after_ping(
323         &mut self,
324         staker_id: AccountId,
325         decrease_amount: U128,
326         beneficiary: AccountId,
327     ) -> PromiseOrValue<Option<StakingChangeResult>> {
328         match env::promise_result(0) {
329             PromiseResult::NotReady => unreachable!(),
330             PromiseResult::Failed => {
331                 emit_callback_failed_event();
332                 PromiseOrValue::Value(None)
```

```
333             }
334             PromiseResult::Successful(_) => {
335                 let mut staker = self.internal_get_staker_or_panic(&staker_id);
336                 let staking_pool = self.internal_get_staking_pool_by_staker_or_panic(&staker_id);
337
338
339                 let decrease_shares = staking_pool.calculate_decrease_shares(decrease_amount.0);
340                 let receive_amount =
341                     staking_pool.staked_amount_from_shares_balance_rounded_up(decrease_shares);
342                 staker.shares = staker
343                     .shares
344                     .checked_sub(decrease_shares)
345                     .expect("Failed decrease shares in staker.");
346
347
348                 self.internal_save_staker(&staker_id, &staker);
349
350
351                 ext_staking_pool::ext(staking_pool.pool_id.clone())
352                     .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_UNSTAKE))
353                     .unstake(receive_amount.into())
354                     .function_call(
355                         "get_account_staked_balance".to_string(),
356                         json!({ "account_id": env::current_account_id() })
357                             .to_string()
358                             .into_bytes(),
359                         0,
360                         Gas::ONE_TERA.mul(TGAS_FOR_GET_ACCOUNT_STAKED_BALANCE),
361                     )
362                     .then(
363                         Self::ext(env::current_account_id())
364                             .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_DECREASE_STAKE_CALL_BACK))
365                             .decrease_stake_callback(
366                                 staker_id,
367                                 decrease_shares.into(),
368                                 receive_amount.into(),
369                                 beneficiary,
370                                 None,
371                             ),
372                     )
373                     .into()
374             }
375         }
376     }
```

Listing 2.34: staking_impl.rs

Impact The last user will not be able to withdraw their full balance.

Suggestion Rounding down when calculating withdrawn amounts from shares.

2.1.14 Potential Panic in Callback Function bond_callback()

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In function `bond()` of contract `RestakingBaseContract`, after invoking the function `bond()` of the contract `consumer_chain.pos_account_id` through a cross-contract call, the subsequent step is to invoke the callback function `bond_callback()` to handle the returned promise result. However, when a successful result is returned, there is a possibility of encountering a panic during the execution of the logic to update the staker bonding state (e.g., `assert!(self.unbonding_unlock_time <= env::block_timestamp())`. This can lead to inconsistencies in the state of both contracts.

```
189   fn bond(&mut self, consumer_chain_id: ConsumerChainId, key: String) -> PromiseOrValue<bool> {
190       self.assert_attached_storage_fee();
191
192
193       let staker_id = env::predecessor_account_id();
194       let consumer_chain = self.internal_get_consumer_chain_or_panic(&consumer_chain_id);
195
196
197       self.ping(Option::None)
198           .then(
199               ext_consumer_chain_pos::ext(consumer_chain.pos_account_id)
200                   .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_CHANGE_KEY))
201                   .bond(staker_id.clone(), key.clone()),
202           )
203           .then(
204               Self::ext(env::current_account_id())
205                   .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_BOND_CALLBACK))
206                   .bond_callback(consumer_chain_id, key, staker_id),
207           )
208           .into()
209   }
```

Listing 2.35: restaking_impl.rs

```
249   #[private]
250   fn bond_callback(
251       &mut self,
252       consumer_chain_id: ConsumerChainId,
253       key: String,
254       staker_id: AccountId,
255   ) -> PromiseOrValue<bool> {
256       match env::promise_result(0) {
257           PromiseResult::NotReady => unreachable!(),
258           PromiseResult::Failed => {
259               emit_callback_failed_event();
260               PromiseOrValue::Value(false)
261           }
262           PromiseResult::Successful(_) => {
263               let mut staker = self.internal_get_staker_or_panic(&staker_id);
```

```
264                 let mut consumer_chain =
265                     self.internal_get_consumer_chain_or_panic(&consumer_chain_id);
266
267
268                 staker.bond(&consumer_chain_id, consumer_chain.unbonding_period);
269                 consumer_chain.bond(&staker_id);
270
271
272                 self.internal_save_staker(&staker_id, &staker);
273                 self.internal_save_consumer_chain(&consumer_chain_id, &consumer_chain);
274
275
276                 Event::StakerBond {
277                     staker_id: &staker_id,
278                     consumer_chain_id: &consumer_chain_id,
279                     key: &key,
280                 }
281                 .emit();
282                 PromiseOrValue::Value(true)
283             }
284         }
285     }
```

Listing 2.36: restaking_impl.rs

Impact The state is not correctly updated due to panic when handling the successful promise result.

Suggestion Avoid panic in successful promise results.

2.1.15 Ineffective Lock on Important Functions

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the current implementation, several functions implement the lock and unlock logic to prevent status inconsistencies caused by asynchronous operations. However, the locked staking pools can be easily unlocked by invoking the function `ping()` as it does not have any locking logic, while in its callback function `ping_callback()`, function `unlock()` is invoked.

The same behavior also exists in the function `decrease_stake_callback()` of the slash process.

```
773     #[private]
774     fn ping_callback(&mut self, pool_id: PoolId) {
775         match env::promise_result(0) {
776             PromiseResult::NotReady => unreachable!(),
777             PromiseResult::Successful(value) => {
778                 let staked_balance: U128 = near_sdk::serde_json::from_slice(&value).unwrap();
779
780
781                 self.internal_use_staking_pool_or_panic(&pool_id, |staking_pool| {
782                     staking_pool.total_staked_balance = staked_balance.0;
783                     staking_pool.unlock();
```

```
784                 });
785
786
787                 Event::Ping {
788                     pool_id: &pool_id,
789                     new_total_staked_balance: &staked_balance,
790                 }
791                 .emit();
792             }
793             PromiseResult::Failed => {
794                 self.internal_use_staking_pool_or_panic(&pool_id, |staking_pool| {
795                     staking_pool.unlock();
796                 });
797             }
798         }
799     }
```

Listing 2.37: staking_impl.rs

```
460     #[private]
461     fn decrease_stake_callback(
462         &mut self,
463         staker_id: AccountId,
464         decrease_shares: U128,
465         decrease_amount: U128,
466         beneficiary: AccountId,
467         slash_treasury: Option<AccountId>,
468     ) -> PromiseOrValue<Option<StakingChangeResult>> {
469         match env::promise_result(0) {
470             PromiseResult::NotReady => unreachable!(),
471             PromiseResult::Successful(value) => {
472                 let new_total_staked_balance = near_sdk::serde_json::from_slice::<U128>(&value)
473                     .expect("Failed to deserialize in decrease_stake_callback by value.")
474                     .0;
475                 let mut staker = self.internal_get_staker_or_panic(&staker_id);
476                 let selected_pool_id = self.internal_get_staker_selected_pool_or_panic(&staker_id);
477                 let mut staking_pool = self.internal_get_staking_pool_or_panic(&selected_pool_id);
478
479
480                 staking_pool.total_staked_balance = new_total_staked_balance;
481                 staking_pool.decrease_stake(decrease_shares.0, new_total_staked_balance);
482                 staking_pool.unlock();
483
484
485                 let pending_withdrawal = self.internal_create_pending_withdrawal_in_staker(
486                     &mut staker,
487                     beneficiary,
488                     decrease_amount.0,
489                     staking_pool.pool_id.clone(),
490                     true,
491                 );
492                 self.internal_save_staking_pool(&staking_pool);
493                 self.internal_save_staker(&staker_id, &staker);
```

```
494
495
496             let sequence = U64::from(self.next_sequence());
497             Event::StakerDecreaseStake {
498                 staking_pool_info: &(&mut staking_pool).into(),
499                 staker_info: &(&staker).into(),
500                 decrease_stake_amount: &decrease_amount,
501                 decrease_shares: &decrease_shares,
502                 pending_withdrawal: &pending_withdrawal,
503                 sequence: &sequence,
504             }
505             .emit();
506
507
508             PromiseOrValue::Value(Some(StakingChangeResult {
509                 sequence: sequence,
510                 new_total_staked_balance: staking_pool
511                     .staked_amount_from_shares_balance_rounded_down(staker.shares)
512                     .into(),
513                 withdrawal_certificate: None,
514             }))
515         }
516         PromiseResult::Failed => {
517             let selected_pool_id = self.internal_get_staker_selected_pool_or_panic(&staker_id);
518
519
520             match slash_treasury {
521                 Some(treasury_account_id) => {
522                     let mut treasury_account =
523                         self.internal_get_account_or_new(&treasury_account_id);
524                     treasury_account
525                         .save_legacy_shares(decrease_shares.0, selected_pool_id.clone());
526                     self.internal_save_account(&treasury_account_id, &treasury_account);
527                 }
528                 None => {
529                     self.internal_decrease_stake_rollback(&staker_id, decrease_shares.0);
530                 }
531             }
532             self.internal_use_staker_staking_pool_or_panic(&staker_id, |staking_pool| {
533                 staking_pool.unlock();
534             });
535             emit_callback_failed_event();
536             PromiseOrValue::Value(None)
537         }
538     }
539 }
```

Listing 2.38: staking_impl.rs

Impact Locked staking pools can be easily unlocked.

Suggestion Properly implement locking logic.

2.1.16 Lack of Pause Functionality in Function bond()

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The protocol allows the privileged admin to pause most functions for maintenance purposes. However, the function `bond()` can not be paused.

```
189    #[payable]
190    fn bond(&mut self, consumer_chain_id: ConsumerChainId, key: String) -> PromiseOrValue<bool> {
191        self.assert_attached_storage_fee();
192
193        let staker_id = env::predecessor_account_id();
194        let consumer_chain = self.internal_get_consumer_chain_or_panic(&consumer_chain_id);
195
196
197        self.ping(Option::None)
198            .then(
199                ext_consumer_chain_pos::ext(consumer_chain.pos_account_id)
200                    .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_CHANGE_KEY))
201                    .bond(staker_id.clone(), key.clone()),
202            )
203            .then(
204                Self::ext(env::current_account_id())
205                    .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_BOND_CALLBACK))
206                    .bond_callback(consumer_chain_id, key, staker_id),
207            )
208            .into()
209    }
```

Listing 2.39: staking_impl.rs

Impact Users can still invoke function `bond()` when the protocol is paused.

Suggestion Add relative check in function `bond()` to disable users from invoking when the protocol is paused.

2.1.17 Unrefunded Storage Fee of Failed Cross-contract Invocations

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description In contract `LposMarket`, validators can bond themselves to the corresponding `consumer_chain` through the function `bond()`. Since the storage is required for saving the added bonding state, the `RESTAKING_BASE_STORAGE_FEE` is required to invoke the cross-contract function `bond()` in the contract `validator.escrow_id`. However, there is no corresponding callback function to handle the returned result. Specifically, if it fails, the attached storage fee will not be returned.

The above issue also occurs in the function `increase_stake()`, `increase_delegation()`, and `delegate()`.

```
28    fn bond(&mut self, consumer_chain_id: ConsumerChainId, key: String) -> Promise {
29        self.assert_contract_is_running();
30        assert_eq!(env::attached_deposit(), RESTAKING_BASE_STORAGE_FEE);
31        let validator = self.internal_get_validator_or_panic(&env::predecessor_account_id());
32        ext_escrow::ext(validator.escrow_id)
33            .with_attached_deposit(RESTAKING_BASE_STORAGE_FEE)
34            .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_BOND))
35            .bond(consumer_chain_id, key)
36            .into()
37    }
```

Listing 2.40: restaking_impl.rs

```
247    fn bond_callback(
248        &mut self,
249        consumer_chain_id: ConsumerChainId,
250        key: String,
251        staker_id: AccountId,
252    ) -> PromiseOrValue<bool> {
253        let mut consumer_chain = self.internal_get_consumer_chain_or_panic(&consumer_chain_id);
254        let mut staker = self.internal_get_staker_or_panic(&staker_id);
255        match env::promise_result(0) {
256            PromiseResult::NotReady => unreachable!(),
257            PromiseResult::Failed => {
258                consumer_chain.unbond(&staker.staker_id);
259                staker.unbond(&consumer_chain.consumer_chain_id);
260                self.internal_save_consumer_chain(&consumer_chain_id, &consumer_chain);
261                self.internal_save_staker(&staker_id, &staker);
262                emit_callback_failed_event();
263                PromiseOrValue::Value(false)
264            }
265            PromiseResult::Successful(_) => {
266                Event::StakerBond {
267                    staker_id: &staker_id,
268                    consumer_chain_id: &consumer_chain_id,
269                    key: &key,
270                }
271                .emit();
272                PromiseOrValue::Value(true)
273            }
274        }
275    }
```

Listing 2.41: restaking_impl.rs

```
120    fn increase_stake(&mut self, increase_amount: U128) {
121        self.assert_contract_is_running();
122        self.assert_max_gas();
123        let validator_id = env::predecessor_account_id();
124        let validator = self.internal_get_validator_or_panic(&validator_id);
125
126
127        assert!(matches!(validator.status, ValidatorStatus::Staking));
```

```
128         assert!(increase_amount.0 > 0);
129         assert_eq!(STORAGE_FEE + increase_amount.0, env::attached_deposit());
130         self.ping(validator.validator_id).then(
131             Self::ext(env::current_account_id())                    .with_static_gas(Gas::
                ONE_TERA.mul(TGAS_FOR_INCREASE_STAKE_AFTER_PING))
132                 .increase_stake_after_ping(validator_id, increase_amount),
133         );
134     }
```

Listing 2.42: staking_impl.rs

```
572     fn increase_stake_after_ping(&mut self, validator_id: AccountId, increase_amount: U128) {
573         match env::promise_result(0) {
574             PromiseResult::NotReady => unreachable!(),
575             PromiseResult::Failed => {
576                 emit_callback_failed_event();
577                 self.transfer_near(validator_id, increase_amount.0);
578             }
579             PromiseResult::Successful(_) => {
580                 let validator = self.internal_get_validator_or_panic(&validator_id);
581
582
583                 ext_escrow::ext(validator.escrow_id)
584                     .with_attached_deposit(increase_amount.0)
585                     .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_INCREASE_STAKE))
586                     .increase_stake_with_sender(validator_id.clone())
587                     .then(
588                         Self::ext(env::current_account_id())
589                             .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_INCREASE_STAKE_CALL_BACK))
590                             .increase_stake_callback(validator_id, increase_amount),
591                     );
592             }
593         }
594     }
```

Listing 2.43: staking_impl.rs

```
253     fn delegate(&mut self, validator_id: AccountId, delegate_amount: U128) {
254         self.assert_contract_is_running();
255         self.assert_max_gas();
256
257
258         assert!(delegate_amount.0 > 0);
259         assert_eq!(STORAGE_FEE + delegate_amount.0, env::attached_deposit());
260
261
262         let delegator_id = env::predecessor_account_id();
263         let delegator = self.internal_get_delegator_or_insert_new(&delegator_id);
264
265
266         assert!(
267             delegator.share_balance == 0 && delegator.select_validator_id.is_none(),
268             "Failed to delegate, already stake in {:?}.",
```

```
269             delegator.select_validator_id
270         );
271
272
273         assert!(
274             delegate_amount.0 >= self.settings.minimum_delegator_stake_amount,
275             "Failed to stake, attach near({}) less than minimum_delegator_stake_amount.({}) ",
276             env::attached_deposit(),
277             self.settings.minimum_delegator_stake_amount
278         );
279         let validator = self.internal_get_validator_or_panic(&validator_id);
280
281
282         assert!((validator.delegators.len() as u32) <= self.settings.maximum_delegators_limit);
283         assert!(matches!(validator.status, ValidatorStatus::Staking));
284
285
286         self.ping(validator.validator_id).then(
287             Self::ext(env::current_account_id())
288                 .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_INCREASE_STAKE_AFTER_PING))
289                 .delegate_after_ping(delegator_id, validator_id, delegate_amount),
290         );
291     }
```

Listing 2.44: staking_impl.rs

```
288     fn increase_delegation(&mut self, increase_amount: U128) {
289         self.assert_contract_is_running();
290         self.assert_max_gas();
291         assert!(increase_amount.0 > 0);
292         assert_eq!(STORAGE_FEE + increase_amount.0, env::attached_deposit());
293
294
295         let delegator_id = env::predecessor_account_id();
296         let delegator = self.internal_get_delegator_or_panic(&delegator_id);
297         let validator =
298             self.internal_get_validator_or_panic(&delegator.select_validator_id.unwrap());
299
300
301         assert!(matches!(validator.status, ValidatorStatus::Staking));
302
303
304         self.ping(validator.validator_id).then(
305             Self::ext(env::current_account_id())
306                 .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_INCREASE_STAKE_AFTER_PING))
307                 .increase_delegation_after_ping(delegator_id, increase_amount),
308         );
309     }
```

Listing 2.45: staking_impl.rs

```
626     fn increase_delegation_after_ping(&mut self, delegator_id: AccountId, increase_amount: U128) {
627         match env::promise_result(0) {
```

```
628             PromiseResult::NotReady => unreachable!(),
629             PromiseResult::Successful(_) => {
630                 let (_, validator) =
631                     self.internal_get_delegator_and_validator_or_panic(&delegator_id);
632
633
634                 ext_escrow::ext(validator.escrow_id)
635                     .with_attached_deposit(increase_amount.0)
636                     .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_INCREASE_STAKE))
637                     .with_unused_gas_weight(0)
638                     .increase_stake_with_sender(delegator_id.clone())
639                     .then(
640                         Self::ext(env::current_account_id())
641                             .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_INCREASE_STAKE_CALL_BACK))
642                             .increase_delegation_callback(delegator_id, increase_amount),
643                     );
644             }
645             PromiseResult::Failed => {
646                 emit_callback_failed_event();
647                 self.transfer_near(delegator_id, increase_amount.0);
648             }
649         }
650     }
```

Listing 2.46: staking_impl.rs

```
596     fn delegate_after_ping(
597         &mut self,
598         delegator_id: AccountId,
599         validator_id: AccountId,
600         amount: U128,
601     ) {
602         match env::promise_result(0) {
603             PromiseResult::NotReady => unreachable!(),
604             PromiseResult::Failed => {
605                 emit_callback_failed_event();
606                 self.transfer_near(delegator_id, amount.0);
607             }
608             PromiseResult::Successful(_) => {
609                 let validator = self.internal_get_validator_or_panic(&validator_id);
610
611
612                 ext_escrow::ext(validator.escrow_id)
613                     .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_INCREASE_STAKE))
614                     .with_attached_deposit(amount.0)
615                     .with_unused_gas_weight(0)
616                     .increase_stake_with_sender(delegator_id.clone())
617                     .then(
618                         Self::ext(env::current_account_id())
619                             .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_INCREASE_STAKE_CALL_BACK))
620                             .delegate_callback(delegator_id, validator_id, amount),
621                     );
622             }
```

```
623         }
624     }
```

Listing 2.47: staking_impl.rs

Impact In cases where the cross-contract invocation fails, the attached storage fee will not be refunded, resulting in a loss to the user.

Suggestion Implement the corresponding callback function to handle the returned promise result correctly.

Feedback from the Project The team believes that the occurrence of failures in the callback is relatively low, and adding the logic for it would introduce unnecessary complexity.

2.1.18 Incorrect Amount of NEAR Attached in Function change_key()

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `LposMarket`, the function `change_key()` is utilized to modify the access key. This function internally invokes the function `change_key()` of the contract escrow with an attached amount of `ONE_YOCTO NEAR`. However, in contract `escrow`, the function `change_key()` requires the attached `NEAR` must be equal to `STORAGE_FEE`. Otherwise, it will revert (line 97), which means the invocation of the function `change_key()` from `restaking_impl.rs` will never be successfully executed.

```
16    fn change_key(&mut self, consumer_chain_id: ConsumerChainId, key: String) -> Promise {
17        self.assert_contract_is_running();
18        assert_one_yocto();
19        let validator = self.internal_get_validator_or_panic(&env::predecessor_account_id());
20        ext_escrow::ext(validator.escrow_id)
21            .with_attached_deposit(ONE_YOCTO)
22            .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_CHANGE_KEY))
23            .change_key(consumer_chain_id, key)
24            .into()
25    }
```

Listing 2.48: restaking_impl.rs

```
91    fn change_key(
92        &mut self,
93        consumer_chain_id: ConsumerChainId,
94        new_key: String,
95    ) -> PromiseOrValue<bool> {
96        self.assert_lpos_market();
97        assert_attached_storage_fee();
98        ext_escrow::ext(self.restaking_base_contract.clone())
99            .with_attached_deposit(STORAGE_FEE)
100           .change_key(consumer_chain_id, new_key)
101           .into()
102   }
```

Listing 2.49: interface.rs

```
170    pub fn assert_attached_storage_fee() {
171   assert_eq!(
172       env::attached_deposit(),
173       STORAGE_FEE,
174       "Should attach {} near as storage fee.",
175       STORAGE_FEE
176   );
177}
```

Listing 2.50: interface.rs

```
19    pub const STORAGE_FEE: Balance = parse_near!("0.01 near");
```

Listing 2.51: typers.rs

Impact The function `change_key()` is actually disabled due to the incorrect amount of `NEAR` attached while making the cross-contract call.

Suggestion Attach the correct amount of `NEAR`.

2.1.19 Incorrect Gas Setting in the Function bond()

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `RestakingBaseContract`, the function `bond()` is used to bind the validator with a specific `consumer_chain`. However, in the function, the gas used for invoking the function `bond()` of the contract `consumer_chain.pos_account_id` is incorrect. It is set to `TGAS_FOR_CHANGE_KEY` instead of `TGAS_FOR_BOND`.

```
189    fn bond(&mut self, consumer_chain_id: ConsumerChainId, key: String) -> PromiseOrValue<bool> {
190       self.assert_attached_storage_fee();
191
192
193       let staker_id = env::predecessor_account_id();
194       let consumer_chain = self.internal_get_consumer_chain_or_panic(&consumer_chain_id);
195
196
197       self.ping(Option::None)
198           .then(
199               ext_consumer_chain_pos::ext(consumer_chain.pos_account_id)
200                   .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_CHANGE_KEY))
201                   .bond(staker_id.clone(), key.clone()),
202           )
203           .then(
204               Self::ext(env::current_account_id())
205                   .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_BOND_CALLBACK))
206                   .bond_callback(consumer_chain_id, key, staker_id),
207           )
208           .into()
209   }
```

Listing 2.52: restaking_impl.rs

```
11    pub const TGAS_FOR_CHANGE_KEY: u64 = 30;
12    pub const TGAS_FOR_BOND: u64 = 50;
```

Listing 2.53: gas_constants.rs

Impact Incorrect gas setting may lead to the failure of invocation.

Suggestion Replace `TGAS_FOR_CHANGE_KEY` with `TGAS_FOR_BOND`.

2.1.20 Unintended Overpayment of Fees by Contract Account

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In function `stake()` of the contract `LposMarket`, the cross-contract invocation to the function `stake_with_validator()` of the contract escrow will attach two extra `yocto NEAR` in addition to the `NEAR` originally attached by the user. According to the design, the additional `NEAR` should also be paid by the user, rather than the contract's account itself.

```
86    fn stake(&mut self, select_pool: PoolId) {
87       self.assert_contract_is_running();
88       self.assert_max_gas();
89       let validator_id = env::predecessor_account_id();
90       let validator = self.internal_get_validator_or_panic(&validator_id);
91       assert!(matches!(validator.status, ValidatorStatus::Deployed));
92       let stake_amount = env::attached_deposit();
93       assert!(
94           validator.total_share_balance == 0 && validator.select_staking_pool.is_none(),
95           "Failed to stake, already stake in {:?}.",
96           validator.select_staking_pool
97       );
98       assert!(
99           stake_amount >= self.settings.minimum_validator_stake_amount,
100          "Failed to stake, attach near({}) less than minimum_validator_stake_amount.({}) ",
101          env::attached_deposit(),
102          self.settings.minimum_validator_stake_amount
103      );
104      ext_escrow::ext(validator.escrow_id)
105          .with_attached_deposit(add_two_yocto(stake_amount))
106          .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_STAKE))
107          .with_unused_gas_weight(0)
108          .stake_with_validator(select_pool.clone(), validator_id.clone())
109          .then(
110              Self::ext(env::current_account_id())
111                  .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_STAKE_CALL_BACK))
112                  .with_unused_gas_weight(0)
113                  .stake_callback(validator_id, select_pool, stake_amount.into()),
114          );
```

```
115   }
```

Listing 2.54: staking_impl.rs

```
23    pub fn add_two_yocto(amount: u128) -> u128 {
24   amount + ONE_YOCTO + ONE_YOCTO
25}
```

Listing 2.55: util.rs

Impact The contract account may run out of `NEAR` deposited for storage cost with the increasing number of staking operations by users.

Suggestion Charge the extra two `yocto NEAR` from the users.

2.1.21 Failure to Clear State Due to Delayed State Saving

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `clean_validator_state()` of the contract `LposMarket`, the state is cleaned by iterating over each `delegator` in `delegators`, and before cleaning the state of each `delegator`, the function will first check whether the `remaining_gas` is enough. If not, the function will return immediately. However, the updated `validator` in the previous iterations is not saved when returning, leading to no `delegator` being cleaned eventually.

Additionally, since this function has an immutable specified gas limit, if the gas is insufficient, the cleaning of the `validator` state can never be completed.

```
187   fn clean_validator_state(
188       &mut self,
189       validator_id: AccountId,
190   ) -> MultiTxsOperationProcessingResult {
191       self.assert_contract_is_running();
192       let mut validator = self.internal_get_validator_or_panic(&validator_id);
193
194
195       assert!(validator.is_able_clean_state());
196
197
198       let gas_limit = Gas::ONE_TERA.mul(20);
199       let delegators = validator.delegators_shares_in_sequence.keys().collect_vec();
200       for delegator in delegators {
201           if remaining_gas() < gas_limit {
202               return MultiTxsOperationProcessingResult::NeedMoreGas;
203           }
204           let mut shares_in_sequence = validator
205               .delegators_shares_in_sequence
206               .get(&delegator)
207               .unwrap();
208           shares_in_sequence.clear();
209           validator.delegators_shares_in_sequence.remove(&delegator);
```

```
210         }
211
212
213         if remaining_gas() < gas_limit {
214             self.internal_save_validator(&validator);
215             return MultiTxsOperationProcessingResult::NeedMoreGas;
216         }
217
218
219         validator.delegator_set_in_sequence.clear();
220
221
222         if remaining_gas() < gas_limit {
223             self.internal_save_validator(&validator);
224             return MultiTxsOperationProcessingResult::NeedMoreGas;
225         }
226         validator.validator_states_in_sequence.clear();
227         self.internal_save_validator(&validator);
228
229
230         return MultiTxsOperationProcessingResult::Ok;
231     }
```

Listing 2.56: staking_impl.rs

```
23    pub fn add_two_yocto(amount: u128) -> u128 {
24   amount + ONE_YOCTO + ONE_YOCTO
25}
```

Listing 2.57: util.rs

Impact Unable to clean the validator state due to insufficient gas.

Suggestion Invoke `self.internal_save_validator()` to save the updated `validator` before it returns because of insufficient gas.

2.1.22 Potential DoS in internal_slash_in_staker_shares()

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `internal_slash_in_staker_shares()`, user shares are unstaked as a slashing penalty for the corresponding `consumer chain`. The total balance of the `staker` (i.e.,`slash_staker_total_balance`) is calculated from the total shares of the `staker` with upward rounding, which must be larger than the actual total balance of the `staker`. If the slash amount specified by the `consumer chain` is larger than the `slash_staker_total_balance`, then it will try to slash all the balance of the `staker`. However, when calculating the final amount of shares to be unstaked, not only is the already inflated `slash_staker_total_balance` used, but there is also an additional rounding up performed, resulting in an underflow issue.

```
423    pub(crate) fn internal_slash_in_staker_shares(
424        &mut self,
425        slash_staker_id: &StakerId,
426        slash_amount: Balance,
427        treasury: &AccountId,
428    ) -> Balance {
429        let pool_id = self.internal_get_staker_selected_pool_or_panic(slash_staker_id);
430        let staker = self.internal_get_staker_or_panic(&slash_staker_id);
431        let staking_pool = self.internal_get_staking_pool_by_staker_or_panic(&slash_staker_id);
432
433
434        let slash_staker_total_balance =
435            staking_pool.staked_amount_from_shares_balance_rounded_up(staker.shares);
436        let actual_slash_amount = min(slash_staker_total_balance, slash_amount);
437
438
439        let (decrease_shares, receive_amount) =
440            self.internal_decrease_stake(&slash_staker_id, actual_slash_amount);
441
442
443        ext_staking_pool::ext(pool_id)
444            .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_UNSTAKE))
445            .with_attached_deposit(ONE_YOCTO)
446            .unstake(receive_amount.into())
447            .function_call(
448                "get_account_staked_balance".to_string(),
449                json!({ "account_id": env::current_account_id() })
450                    .to_string()
451                    .into_bytes(),
452                0,
453                Gas::ONE_TERA.mul(TGAS_FOR_GET_ACCOUNT_STAKED_BALANCE),
454            )
455            .then(
456                Self::ext(env::current_account_id())
457                    .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_UNSTAKE))
458                    .decrease_stake_callback(
459                        slash_staker_id.clone(),
460                        decrease_shares.into(),
461                        receive_amount.into(),
462                        treasury.clone(),
463                        Some(treasury.clone()),
464                    ),
465            );
466        actual_slash_amount
467    }
```

Listing 2.58: restaking_impl.rs

```
839    pub(crate) fn internal_decrease_stake(
840        &mut self,
841        staker_id: &StakerId,
842        decrease_amount: Balance,
843    ) -> (ShareBalance, Balance) {
```

```
844         assert!(
845             decrease_amount > 0,
846             "Decrease stake amount should be positive"
847         );
848         let mut staker = self.internal_get_staker_or_panic(staker_id);
849         let pool_id = &self.internal_get_staker_selected_pool_or_panic(staker_id);
850         let staking_pool = self.internal_get_staking_pool_or_panic(pool_id);
851
852
853         // Calculate the number of shares required to unstake the given amount.
854         // NOTE: The number of shares the account will pay is rounded up.
855         let num_shares = staking_pool.num_shares_from_staked_amount_rounded_up(decrease_amount);
856         assert!(
857             num_shares > 0,
858             "Invariant violation. The calculated number of stake shares for unstaking should be
                 positive"
859         );
860         assert!(
861             staker.shares >= num_shares,
862             "Not enough staked balance to unstake"
863         );
864
865
866         // Calculating the amount of tokens the account will receive by unstaking the corresponding
867         // number of "stake" shares, rounding up.
868         let receive_amount = staking_pool.staked_amount_from_shares_balance_rounded_up(num_shares);
869         assert!(
870             receive_amount > 0,
871             "Invariant violation. Calculated staked amount must be positive, because stake share
                 price should be at least 1"
872         );
873
874
875         staker.shares -= num_shares;
876
877
878         self.internal_save_staker(staker_id, &staker);
879
880
881         (num_shares, receive_amount)
882     }
```

Listing 2.59: staking_impl.rs

Impact Unable to slash users due to arithmetic underflow.

Suggestion If the value of the final computed slashed shares is larger than the total shares of the staker, slash the total shares of the staker.

2.1.23 Incorrect Penalty Amount in the Slash Process

Severity High

Status Fixed in `Version 2`

Introduced by Version 1

Description In the slashing process, the final penalty amount `receive_amount` is calculated from `actual_slash_amount` with two rounds of upward rounding calculations. When `actual_slash_amount` equals to `slash_amount`, the `receive_amount` will exceed `slash_amount`. Consequently, more funds than the specified `slash_amount` are withdrawn from the corresponding staking pool as a penalty, resulting in a loss to the funds of other users in the `stake_pool`.

```
423    pub(crate) fn internal_slash_in_staker_shares(
424        &mut self,
425        slash_staker_id: &StakerId,
426        slash_amount: Balance,
427        treasury: &AccountId,
428    ) -> Balance {
429        let pool_id = self.internal_get_staker_selected_pool_or_panic(slash_staker_id);
430        let staker = self.internal_get_staker_or_panic(&slash_staker_id);
431        let staking_pool = self.internal_get_staking_pool_by_staker_or_panic(&slash_staker_id);
432
433
434        let slash_staker_total_balance =
435            staking_pool.staked_amount_from_shares_balance_rounded_up(staker.shares);
436        let actual_slash_amount = min(slash_staker_total_balance, slash_amount);
437
438
439        let (decrease_shares, receive_amount) =
440            self.internal_decrease_stake(&slash_staker_id, actual_slash_amount);
441
442
443        ext_staking_pool::ext(pool_id)
444            .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_UNSTAKE))
445            .with_attached_deposit(ONE_YOCTO)
446            .unstake(receive_amount.into())
447            .function_call(
448                "get_account_staked_balance".to_string(),
449                json!({ "account_id": env::current_account_id() })
450                    .to_string()
451                    .into_bytes(),
452                0,
453                Gas::ONE_TERA.mul(TGAS_FOR_GET_ACCOUNT_STAKED_BALANCE),
454            )
455            .then(
456                Self::ext(env::current_account_id())
457                    .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_UNSTAKE))
458                    .decrease_stake_callback(
459                        slash_staker_id.clone(),
460                        decrease_shares.into(),
461                        receive_amount.into(),
462                        treasury.clone(),
463                        Some(treasury.clone()),
464                    ),
465            );
466        actual_slash_amount
467    }
```

Listing 2.60: restaking_impl.rs

```
839    pub(crate) fn internal_decrease_stake(
840        &mut self,
841        staker_id: &StakerId,
842        decrease_amount: Balance,
843    ) -> (ShareBalance, Balance) {
844        assert!(
845            decrease_amount > 0,
846            "Decrease stake amount should be positive"
847        );
848        let mut staker = self.internal_get_staker_or_panic(staker_id);
849        let pool_id = &self.internal_get_staker_selected_pool_or_panic(staker_id);
850        let staking_pool = self.internal_get_staking_pool_or_panic(pool_id);
851
852
853        // Calculate the number of shares required to unstake the given amount.
854        // NOTE: The number of shares the account will pay is rounded up.
855        let num_shares = staking_pool.num_shares_from_staked_amount_rounded_up(decrease_amount);
856        assert!(
857            num_shares > 0,
858            "Invariant violation. The calculated number of stake shares for unstaking should be
                positive"
859        );
860        assert!(
861            staker.shares >= num_shares,
862            "Not enough staked balance to unstake"
863        );
864
865
866        // Calculating the amount of tokens the account will receive by unstaking the corresponding
867        // number of "stake" shares, rounding up.
868        let receive_amount = staking_pool.staked_amount_from_shares_balance_rounded_up(num_shares);
869        assert!(
870            receive_amount > 0,
871            "Invariant violation. Calculated staked amount must be positive, because stake share
                price should be at least 1"
872        );
873
874
875        staker.shares -= num_shares;
876
877
878        self.internal_save_staker(staker_id, &staker);
879
880
881        (num_shares, receive_amount)
882    }
```

Listing 2.61: staking_impl.rs

Impact Users who have not been slashed still suffer a loss of funds by slashing.

Suggestion Rounding down when calculating the `receive_amount` to be slashed.

2.1.24 Unlimited Delay in Asset Withdrawal Due to Continuous Invocation of decease_stake()

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `RestakingBaseContract`, the function `decrease_stake()` serves to unstake the `NEAR` from the selected staking pool and generates a `pending_withdrawal` for users. The unstaked `NEAR` can only be withdrawn if the current time exceeds the lock time specified when generating the `pending_withdrawal`.

However, each invocation of the function `decrease_stake()`, the internal function `decrease_stake()` in `staking_pool.rs` will reset the `staking_pool's unlock_epoch` to `env::epoch_height() + NUM_EPOCHS_TO_UNLOCK`. This value is used to check whether the unstaked assets are available to withdraw in function `withdraw()`. Consequently, if the function `decrease_stake()` is continuously called, the available time for users to withdraw their assets will be indefinitely extended.

```
96    fn decrease_stake(
97        &mut self,
98        decrease_amount: U128,
99        beneficiary: Option<AccountId>,
100   ) -> PromiseOrValue<Option<StakingChangeResult>> {
101       self.assert_contract_is_running();
102       self.assert_attached_storage_fee();
103       assert!(decrease_amount.0 > 0, "The decrease amount should gt 0");
104
105
106       let staker_id = env::predecessor_account_id();
107       self.internal_use_staker_staking_pool_or_panic(&staker_id, |staking_pool| {
108           staking_pool.lock()
109       });
110
111
112       return self
113           .ping(Option::None)
114           .then(
115               Self::ext(env::current_account_id())
116                   .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_DECREASE_STAKE_AFTER_PING))
117                   .decrease_stake_after_ping(
118                       staker_id,
119                       decrease_amount,
120                       beneficiary.unwrap_or(env::predecessor_account_id()),
121                   ),
122           )
123           .into();
124   }
```

Listing 2.62: staking_impl.rs

```rust
    fn decrease_stake_callback(
        &mut self,
        staker_id: AccountId,
        decrease_shares: U128,
        decrease_amount: U128,
        beneficiary: AccountId,
        slash_treasury: Option<AccountId>,
    ) -> PromiseOrValue<Option<StakingChangeResult>> {
        match env::promise_result(0) {
            PromiseResult::NotReady => unreachable!(),
            PromiseResult::Successful(value) => {
                let new_total_staked_balance = near_sdk::serde_json::from_slice::<U128>(&value)
                    .expect("Failed to deserialize in decrease_stake_callback by value.")
                    .0;
                let mut staker = self.internal_get_staker_or_panic(&staker_id);
                let selected_pool_id = self.internal_get_staker_selected_pool_or_panic(&staker_id);
                let mut staking_pool = self.internal_get_staking_pool_or_panic(&selected_pool_id);


                staking_pool.total_staked_balance = new_total_staked_balance;
                staking_pool.decrease_stake(decrease_shares.0, new_total_staked_balance);
                staking_pool.unlock();


                let pending_withdrawal = self.internal_create_pending_withdrawal_in_staker(
                    &mut staker,
                    beneficiary,
                    decrease_amount.0,
                    staking_pool.pool_id.clone(),
                    true,
                );
                self.internal_save_staking_pool(&staking_pool);
                self.internal_save_staker(&staker_id, &staker);


                let sequence = U64::from(self.next_sequence());
                Event::StakerDecreaseStake {
                    staking_pool_info: &(&mut staking_pool).into(),
                    staker_info: &(&staker).into(),
                    decrease_stake_amount: &decrease_amount,
                    decrease_shares: &decrease_shares,
                    pending_withdrawal: &pending_withdrawal,
                    sequence: &sequence,
                }
                .emit();


                PromiseOrValue::Value(Some(StakingChangeResult {
                    sequence: sequence,
                    new_total_staked_balance: staking_pool
                        .staked_amount_from_shares_balance_rounded_down(staker.shares)
                        .into(),
```

```
                withdrawal_certificate: None,
            }))
        }
        PromiseResult::Failed => {
            let selected_pool_id = self.internal_get_staker_selected_pool_or_panic(&staker_id);


            match slash_treasury {
                Some(treasury_account_id) => {
                    let mut treasury_account =
                        self.internal_get_account_or_new(&treasury_account_id);
                    treasury_account
                        .save_legacy_shares(decrease_shares.0, selected_pool_id.clone());
                    self.internal_save_account(&treasury_account_id, &treasury_account);
                }
                None => {
                    self.internal_decrease_stake_rollback(&staker_id, decrease_shares.0);
                }
            }
            self.internal_use_staker_staking_pool_or_panic(&staker_id, |staking_pool| {
                staking_pool.unlock();
            });
            emit_callback_failed_event();
            PromiseOrValue::Value(None)
        }
    }
}
```

Listing 2.63: staking_impl.rs

```
 pub fn decrease_stake(
    &mut self,
    decrease_shares: ShareBalance,
    new_total_staked_balance: Balance,
) {
    self.total_share_balance -= decrease_shares;
    self.total_staked_balance = new_total_staked_balance;
    self.unlock_epoch = env::epoch_height() + NUM_EPOCHS_TO_UNLOCK;
}
```

Listing 2.64: staking_pool.rs

```
 fn withdraw(&mut self, staker: AccountId, id: WithdrawalCertificate) -> PromiseOrValue<U128> {
    self.assert_contract_is_running();
    let pending_withdrawal = self.internal_use_account(&staker, |account| {
        account.pending_withdrawals.remove(&id).unwrap()
    });
    let staking_pool = self.internal_get_staking_pool_or_panic(&pending_withdrawal.pool_id);
    assert!(
        pending_withdrawal.is_withdrawable() && staking_pool.is_withdrawable(),
        "unlock timestamp:{}, unlock epoch:{}, current timestamp:{}, current epoch: {}, staking
            pool unlock epoch: {}",
        pending_withdrawal.unlock_time,
```

```
164             pending_withdrawal.unlock_epoch,
165             env::block_timestamp(),
166             env::epoch_height(),
167             staking_pool.unlock_epoch
168         );
169         assert!(
170             pending_withdrawal.allow_other_withdraw
171                 || env::predecessor_account_id().eq(&pending_withdrawal.beneficiary)
172         );
173
174
175         ext_staking_pool::ext(pending_withdrawal.pool_id.clone())
176             .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_WITHDRAW))
177             .withdraw(pending_withdrawal.amount.into())
178             .then(
179                 Self::ext(env::current_account_id())
180                     .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_SINGLE_WITHDRAW_CALLBACK))
181                     .withdraw_callback(staker, pending_withdrawal),
182             )
183             .into()
184     }
```

Listing 2.65: staking_impl.rs

```
95      pub fn is_withdrawable(&self) -> bool {
96          self.unlock_epoch <= env::epoch_height()
97      }
```

Listing 2.66: staking_pool.rs

Impact Users are unable to withdraw their unstaked NEAR as scheduled.

Suggestion Each user's `pending_withdrawal` lock time should be independent and not affected by each other's operation.

2.1.25 Unrefunded STORAGE_FEE of Released Storage

Severity Low

Status Confirmed

Introduced by Version 1

Description In contract `LposMarket`, the function `unbond()` removes the binding relationship between the `validator` and the `consumer_chain` by deleting the insert record of the `validator` and `consumer_chain_id`, which releases the storage. Since the function `bond()` charged a certain amount of storage fee before, when the function `unbond()` is successfully executed, the charged fees should be returned. However, there is no refunding logic in the current implementation.

The above issue also occurs in the function `undelegate()` in `staking_imple.rs`.

```
114     fn unbond(&mut self, consumer_chain_id: ConsumerChainId) -> Promise {
115         self.assert_contract_is_running();
116         assert_one_yocto();
117         let validator = self.internal_get_validator_or_panic(&env::predecessor_account_id());
```

```
118         ext_escrow::ext(validator.escrow_id)
119             .with_attached_deposit(ONE_YOCTO)
120             .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_UNBOND))
121             .unbond(consumer_chain_id)
122             .into()
123     }
```

Listing 2.67: restaking_impl.rs

```
231     fn unbond(&mut self, consumer_chain_id: ConsumerChainId) {
232         assert_one_yocto();
233         let staker_id = env::predecessor_account_id();
234         self.internal_use_staker_or_panic(&staker_id, |staker| staker.unbond(&consumer_chain_id));
235         self.internal_use_consumer_chain_or_panic(&consumer_chain_id, |consumer_chain| {
236             consumer_chain.unbond(&staker_id)
237         });
238         Event::StakerUnbond {
239             staker_id: &staker_id,
240             consumer_chain_id: &consumer_chain_id,
241         }
242         .emit();
243     }
```

Listing 2.68: restaking_impl.rs

```
254     fn delegate(&mut self, validator_id: AccountId, delegate_amount: U128) {
255         self.assert_contract_is_running();
256         self.assert_max_gas();
257
258
259         assert!(delegate_amount.0 > 0);
260         assert_eq!(STORAGE_FEE + delegate_amount.0, env::attached_deposit());
261
262
263         let delegator_id = env::predecessor_account_id();
264         let delegator = self.internal_get_delegator_or_insert_new(&delegator_id);
265
266
267         assert!(
268             delegator.share_balance == 0 && delegator.select_validator_id.is_none(),
269             "Failed to delegate, already stake in {:?}.",
270             delegator.select_validator_id
271         );
272
273
274         assert!(
275             delegate_amount.0 >= self.settings.minimum_delegator_stake_amount,
276             "Failed to stake, attach near({}) less than minimum_delegator_stake_amount.({}) ",
277             env::attached_deposit(),
278             self.settings.minimum_delegator_stake_amount
279         );
280         let validator = self.internal_get_validator_or_panic(&validator_id);
281
```

```
282
283     assert!((validator.delegators.len() as u32) <= self.settings.maximum_delegators_limit);
284     assert!(matches!(validator.status, ValidatorStatus::Staking));
285
286
287     self.ping(validator.validator_id).then(
288         Self::ext(env::current_account_id())
289             .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_INCREASE_STAKE_AFTER_PING))
290             .delegate_after_ping(delegator_id, validator_id, delegate_amount),
291     );
292 }
```

Listing 2.69: staking_impl.rs

```
327  fn undelegate(&mut self) {
328     self.assert_contract_is_running();
329     self.assert_max_gas();
330     assert_eq!(env::attached_deposit(), STORAGE_FEE);
331     let delegator_id = env::predecessor_account_id();
332     let delegator = self.internal_get_delegator_or_panic(&delegator_id);
333     let validator =
334         self.internal_get_validator_or_panic(&delegator.select_validator_id.unwrap());
335     assert!(matches!(validator.status, ValidatorStatus::Staking));
336     self.ping(validator.validator_id.clone()).then(
337         Self::ext(env::current_account_id())
338             .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_DECREASE_STAKE_AFTER_PING))
339             .undelegate_after_ping(delegator_id),
340     );
341 }
```

Listing 2.70: staking_impl.rs

```
765  fn undelegate_after_ping(&mut self, delegator_id: AccountId) {
766     match env::promise_result(0) {
767         PromiseResult::NotReady => unreachable!(),
768         PromiseResult::Successful(_) => {
769             let (mut delegator, validator) =
770                 self.internal_get_delegator_and_validator_or_panic(&delegator_id);
771
772
773             let decrease_shares = delegator.share_balance;
774             delegator.share_balance -= decrease_shares;
775             let near_balance = validator.calculate_near_balance(decrease_shares);
776
777
778             self.internal_save_delegator(&delegator);
779
780
781             ext_escrow::ext(validator.escrow_id)
782                 .with_attached_deposit(RESTAKING_BASE_STORAGE_FEE)
783                 .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_DECREASE_STAKE))
784                 .with_unused_gas_weight(0)
785                 .decrease_stake(near_balance.into(), Some(delegator_id.clone()))
```

```
786                 .then(
787                     Self::ext(env::current_account_id())
788                         .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_DECREASE_STAKE_CALL_BACK))
789                         .undelegate_callback(
790                             delegator_id.clone(),
791                             validator.validator_id,
792                             decrease_shares.into(),
793                             near_balance.into(),
794                         ),
795                 );
796         }
797         PromiseResult::Failed => {
798             emit_callback_failed_event();
799         }
800     }
801 }
```

Listing 2.71: staking_impl.rs

```
327 fn undelegate_callback(
328     &mut self,
329     delegator_id: AccountId,
330     validator_id: AccountId,
331     decreased_shares: U128,
332     undelegate_amount: U128,
333 ) {
334     match env::promise_result(0) {
335         PromiseResult::NotReady => unreachable!(),
336         PromiseResult::Successful(result) => {
337             if let Some(staking_change_result) =
338                 near_sdk::serde_json::from_slice::<Option<StakingChangeResult>>(&result)
339                     .unwrap()
340             {
341                 let mut delegator = self.internal_get_delegator_or_panic(&delegator_id);
342                 let mut validator = self.internal_get_validator_or_panic(&validator_id);
343
344
345                 let sequence = staking_change_result.sequence;
346
347
348                 delegator.select_validator_id = None;
349                 validator.undelegate(
350                     delegator_id.clone(),
351                     decreased_shares.0,
352                     sequence.0,
353                     staking_change_result.new_total_staked_balance.0,
354                 );
355                 self.internal_save_delegator(&delegator);
356                 self.internal_save_validator(&validator);
357
358
359                 Event::Undelegate {
360                     validator_info: &validator.into(),
```

```
                    delegator_info: &delegator.into(),
                    undelegate_amount: &undelegate_amount,
                    undelegate_shares: &decreased_shares,
                    sequence: &sequence,
                }
                .emit();
            } else {
                // rollback decrease
                self.internal_use_delegator_or_panic(&delegator_id, |delegator| {
                    delegator.share_balance += decreased_shares.0;
                });
                emit_callback_failed_event();
            }
        }
        PromiseResult::Failed => {
            // rollback decrease
            self.internal_use_delegator_or_panic(&delegator_id, |delegator| {
                delegator.share_balance += decreased_shares.0;
            });
            emit_callback_failed_event();
        }
    }
}
```

Listing 2.72: staking_impl.rs

Impact Storage fees will not be returned when users' operations release storage.

Suggestion Refund the storage fee once the mentioned operations above executed successfully.

Feedback from the Project Because implementing the storage fee refund would significantly increase the complexity of the contract implementation, the team prefers not to add this additional logic.

2.1.26 No Storage Fee Charged in Function sync_consumer_chain_pos()

Severity Low

Status Confirmed

Introduced by `Version 1`

Description Function `sync_consumer_chain_pos()` is used to synchronize and update the state of `consumer_chains` between the contracts `RestakingBaseContract` and `lpos-market`. If a new consumer chain is registered in `RestakingBaseContract`, the function will also insert the corresponding data into the storage of `LposMarket`, which consumes a certain amount of storage. However, the corresponding storage fee is not charged in this case.

```
fn sync_consumer_chain_pos(&mut self, consumer_chain_id: ConsumerChainId) -> Promise {
    self.assert_contract_is_running();
    ext_restaking_base::ext(self.settings.restaking_base_contract.clone())
        .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_VIEW))
        .get_consumer_chain(consumer_chain_id)
        .then(
            Self::ext(env::current_account_id())
                .with_static_gas(Gas::ONE_TERA.mul(20))
```

```
59                  .sync_consumer_chain_pos_callback(),
60          )
61  }
```

Listing 2.73: restaking_impl.rs

```
256  #[private]
257  pub fn sync_consumer_chain_pos_callback(
258      &mut self,
259      #[callback] consumer_chain_info: Option<ConsumerChainInfo>,
260  ) {
261      if let Some(consumer_chain_info_unwrap) = consumer_chain_info {
262          let mut consumer_chain = self
263              .consumer_chains
264              .get(&consumer_chain_info_unwrap.consumer_chain_id)
265              .unwrap_or(ConsumerChain::new(
266                  consumer_chain_info_unwrap.consumer_chain_id.clone(),
267                  consumer_chain_info_unwrap.pos_account_id.clone(),
268              ));
269          consumer_chain.anchor_id = consumer_chain_info_unwrap.pos_account_id;
270
271
272          self.consumer_chains.insert(
273              &consumer_chain_info_unwrap.consumer_chain_id,
274              &consumer_chain,
275          );
276      }
277  }
```

Listing 2.74: restaking_impl.rs

Impact The contract `LposMarket` will bear additional storage fee.

Suggestion Add storage fee requirement in function `sync_consumer_chain_pos()`.

Feedback from the Project Because the `Consumer Chain` need to pay fee when register in `restakingBase`. So it acceptable to cover the cost of storage in this interface.

2.1.27 Unlimited Withdrawn with Reused UnstakeBatchId

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `withdraw_unstake_batch()` in the contract `RestakingBaseContract` is designed to withdraw `NEAR` from the staking pool when the user's submitted unstaking request is withdrawable. It reads the corresponding unstaking request's status based on the parameter `unstake_batch_id` to determine whether the withdrawal should be executed. However, due to the absence of necessary checks, a valid withdrawable `unstake_batch_id` can be used repeatedly to withdraw the funds from the staking pool. The funds in the staking pool can be drained and thereby preventing other users from withdrawing theirs.

```
155  fn withdraw_unstake_batch(&mut self, pool_id: PoolId, unstake_batch_id: UnstakeBatchId) {
156      self.assert_contract_is_running();
```

```
157         let submitted_unstake_batch =
158             self.internal_use_staking_pool_or_panic(&pool_id, |staking_pool| {
159                 assert!(staking_pool.is_unstake_batch_withdrawable(&unstake_batch_id));
160
161
162                 staking_pool.lock();
163                 staking_pool
164                     .submitted_unstake_batches
165                     .get(&unstake_batch_id)
166                     .unwrap()
167             });
168
169
170         ext_staking_pool::ext(pool_id.clone())
171             .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_WITHDRAW))
172             .with_unused_gas_weight(0)
173             .withdraw(submitted_unstake_batch.total_unstake_amount.into())
174             .then(
175                 Self::ext(current_account_id())
176                     .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_WITHDRAW_UNSTAKE_BATCH_CALLBACK))
177                     .with_unused_gas_weight(0)
178                     .withdraw_unstake_batch_callback(pool_id.clone(), unstake_batch_id),
179             );
180     }
```

Listing 2.75: staking_impl.rs

Impact Funds in the staking pool can be drained by repeatedly withdrawing with the same `unstake_batch_id`.

Suggestion If the corresponding funds of the `unstake_batch_id` are withdrawn, fail the invocation.

2.1.28 Potential Panic in Callback Function stake_after_check_whitelisted()

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description When handling the successful result in the callback function `stake_after_check_whitelisted()`, the execution of `staking_pool.lock()` may panic if the staking pool is already locked. In this case, the `NEAR` deposited by the user will not be refunded, resulting in a loss of the user.

```
688     #[payable]
689     #[private]
690     fn stake_after_check_whitelisted(
691         &mut self,
692         staker_id: AccountId,
693         pool_id: PoolId,
694     ) -> PromiseOrValue<Option<StakingChangeResult>> {
695         match env::promise_result(0) {
696             PromiseResult::NotReady => unreachable!(),
697             PromiseResult::Successful(value) => {
698                 let whitelisted = near_sdk::serde_json::from_slice::<bool>(&value)
699                     .expect("Failed to deserialize in increase_stake_callback by value.");
```

```
700
701
702             if !whitelisted {
703                 log!("Failed to select pool, {} is not whitelisted.", pool_id);
704                 self.transfer_near(staker_id, env::attached_deposit());
705                 return PromiseOrValue::Value(None);
706             }
707
708
709             if !self.staking_pools.get(&pool_id).is_some() {
710                 self.internal_save_staking_pool(&StakingPool::new(pool_id.clone()));
711                 Event::SaveStakingPool { pool_id: &pool_id }.emit();
712             }
713
714
715             self.internal_use_staking_pool_or_panic(&pool_id, |staking_pool| {
716                 staking_pool.lock()
717             });
718
719
720             self.ping(Some(pool_id.clone()))
721                 .then(
722                     Self::ext(env::current_account_id())
723                         .with_attached_deposit(env::attached_deposit())
724                         .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_INCREASE_STAKE_AFTER_PING))
725                         .stake_after_ping(staker_id, pool_id.clone()),
726                 )
727                 .into()
728         }
729         PromiseResult::Failed => {
730             self.transfer_near(staker_id, env::attached_deposit());
731             emit_callback_failed_event();
732             return PromiseOrValue::Value(None);
733         }
734     }
735 }
```

Listing 2.76: staking_impl.rs

Impact If users attempt to stake their `NEAR` tokens while the staking pool is locked, they will lose their `NEARs`.

Suggestion Check if the staking pool is locked, if so, return the attached `NEAR` tokens instead of reverting.

2.1.29 Lack of Storage Fee Charge

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description In the function `submit_unstake_batch()` of the contract `RestakingBaseContract`, it will insert new `SubmittedUnstakeBatch` into `submitted_unstake_batches` if the newly created unstaking request is not submitted. It will consume a certain amount of storage, but the corresponding storage fee is not charged.

The same issue also exists in the function `stake()` of contract `LposMarket`.

```
180    fn submit_unstake_batch(&mut self, pool_id: PoolId) {
181        self.assert_contract_is_running();
182        let mut staking_pool = self.internal_get_staking_pool_or_panic(&pool_id);
183        assert!(staking_pool.is_able_submit_unstake_batch());
184
185
186        staking_pool.lock();
187
188
189        self.internal_save_staking_pool(&staking_pool);
190
191
192        ext_staking_pool::ext(pool_id.clone())
193            .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_UNSTAKE))
194            .with_unused_gas_weight(0)
195            .unstake(staking_pool.batched_unstake_amount.into())
196            .then(
197                Self::ext(current_account_id())
198                    .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_UNSTAKE_BATCH_CALLBACK))
199                    .with_unused_gas_weight(0)
200                    .submit_unstake_batch_callback(pool_id),
201            );
202    }
```

Listing 2.77: staking_impl.rs

```
733    #[private]
734    fn submit_unstake_batch_callback(&mut self, pool_id: PoolId) {
735        match env::promise_result(0) {
736            PromiseResult::NotReady => unreachable!(),
737            PromiseResult::Successful(_) => {
738                let mut staking_pool = self.internal_get_staking_pool_or_panic(&pool_id);
739
740
741                let submitted_unstake_batch = staking_pool.submit_unstake();
742                staking_pool.unlock();
743                self.internal_save_staking_pool(&staking_pool);
744
745
746                Event::SubmitUnstakeBatch {
747                    submitted_unstake_batch: &submitted_unstake_batch,
748                    staking_pool: &staking_pool.into(),
749                }
750                .emit();
751            }
752            PromiseResult::Failed => {
753                self.internal_use_staking_pool_or_panic(&pool_id, |staking_pool| {
754                    staking_pool.unlock();
755                });
756                emit_callback_failed_event();
757            }
758        }
```

```
759     }
```

Listing 2.78: staking_impl.rs

```
202     pub fn submit_unstake(&mut self) -> SubmittedUnstakeBatch {
203         let submitted_unstake_batch = SubmittedUnstakeBatch {
204             unstake_batch_id: self.current_unstake_batch_id,
205             submit_unstake_epoch: env::epoch_height(),
206             total_unstake_amount: self.batched_unstake_amount,
207             claimed_amount: 0,
208             is_withdrawn: false,
209         };
210         self.submitted_unstake_batches.insert(
211             &self.current_unstake_batch_id,
212             &SubmittedUnstakeBatch {
213                 unstake_batch_id: self.current_unstake_batch_id,
214                 submit_unstake_epoch: env::epoch_height(),
215                 total_unstake_amount: self.batched_unstake_amount,
216                 claimed_amount: 0,
217                 is_withdrawn: false,
218             },
219         );
220
221
222         self.last_unstake_epoch = env::epoch_height();
223         self.last_unstake_batch_id = Some(self.current_unstake_batch_id.clone());
224         self.current_unstake_batch_id = (self.current_unstake_batch_id.0 + 1).into();
225         self.batched_unstake_amount = 0;
226
227
228         self.unlock_epoch = env::epoch_height() + NUM_EPOCHS_TO_UNLOCK;
229
230
231         submitted_unstake_batch
232     }
```

Listing 2.79: staking_pool.rs

```
84      #[payable]
85      fn stake(&mut self, select_pool: PoolId) {
86          self.assert_contract_is_running();
87          self.assert_max_gas();
88          let validator_id = env::predecessor_account_id();
89          let validator = self.internal_get_validator_or_panic(&validator_id);
90          assert!(matches!(validator.status, ValidatorStatus::Deployed));
91          let stake_amount = env::attached_deposit();
92
93
94          assert!(
95              validator.total_share_balance == 0 && validator.select_staking_pool.is_none(),
96              "Failed to stake, already stake in {:?}.",
97              validator.select_staking_pool
98          );
```

```
 99        assert!(
100            stake_amount >= self.settings.minimum_validator_stake_amount,
101            "Failed to stake, attach near({}) less than minimum_validator_stake_amount.({}) ",
102            env::attached_deposit(),
103            self.settings.minimum_validator_stake_amount
104        );
105
106
107        ext_staking_pool::ext(select_pool.clone())
108            .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_VIEW))
109            .with_unused_gas_weight(0)
110            .get_owner_id()
111            .then(
112                Self::ext(env::current_account_id())
113                    .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_CHECK_POOL_THEN_STAKE))
114                    .with_unused_gas_weight(1)
115                    .check_staking_pool_then_stake(
116                        validator_id,
117                        validator.escrow_id.clone(),
118                        stake_amount.into(),
119                        select_pool,
120                    ),
121            );
122    }
```

Listing 2.80: staking_impl.rs

```
581    #[private]
582    fn check_staking_pool_then_stake(
583        &mut self,
584        validator_id: AccountId,
585        validator_escrow: AccountId,
586        stake_amount: U128,
587        select_pool_id: PoolId,
588    ) {
589        match env::promise_result(0) {
590            PromiseResult::NotReady => unreachable!(),
591            PromiseResult::Successful(value) => {
592                if let Ok(staking_pool_owner_id) = from_slice::<AccountId>(&value) {
593                    let staking_pool =
594                        self.staking_pools
595                            .get(&select_pool_id)
596                            .unwrap_or(StakingPool::new(
597                                select_pool_id.clone(),
598                                staking_pool_owner_id.clone(),
599                            ));
600                    self.internal_save_staking_pool(&staking_pool);
601                    if staking_pool.is_allow_stake(&validator_id) {
602                        ext_escrow::ext(validator_escrow)
603                            .with_attached_deposit(stake_amount.0)
604                            .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_STAKE))
605                            .with_unused_gas_weight(0)
606                            .stake_with_validator(select_pool_id.clone(), validator_id.clone())
```

```
607                         .then(
608                             Self::ext(env::current_account_id())
609                                 .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_STAKE_CALL_BACK))
610                                 .with_unused_gas_weight(0)
611                                 .stake_callback(
612                                     validator_id,
613                                     select_pool_id,
614                                     stake_amount.into(),
615                                 ),
616                         );
617                 } else {
618                     // if staking pool not allow to stake, should refund
619                     emit_callback_failed_event();
620                     self.transfer_near(validator_id, stake_amount.0);
621                 }
622             } else {
623                 emit_callback_failed_event();
624                 self.transfer_near(validator_id, stake_amount.0);
625             }
626         }
627
628
629         PromiseResult::Failed => {
630             emit_callback_failed_event();
631             self.transfer_near(validator_id, stake_amount.0);
632         }
633     }
634 }
```

Listing 2.81: staking_impl.rs

Impact The contract `LposMarket` and `RestakingBaseContract` will bear additional storage fees.

Suggestion Add storage fee checks in functions `submit_unstake_batch()` and `stake()`.

Feedback from the Project The function `is_able_submit_unstake_batch` have checked that `batched_unstake_amount` should greater than zero. It means at least a `PendingWithdrawal` have created. Team believes storage fee for `PendingWIthdrawal` can cover `SubmittedUnstakeBatch`.

2.1.30 Panic in Callback Function stake_callback()

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the processing of a failed result within the callback function `stake_callback()`, the status of the `staker` and corresponding `stake pool` will be rolled back. Specifically, the `select_staking_pool` of the `staker` will be set as `None`, and the `stake pool` will be unlocked. However, when selecting the `stake pool` to unlock, the staker's `select_staking_pool` with a value of `None` is used. This would cause the function `internal_use_staker_staking_pool_or_panic()` to panic directly, leading to the failure of all state rollbacks. Besides, the deposited `NEAR` tokens would not be refunded.

```
568     #[private]
569     fn stake_callback(
570         &mut self,
571         staker_id: AccountId,
572         stake_amount: U128,
573         pool_id: PoolId,
574     ) -> PromiseOrValue<Option<StakingChangeResult>> {
575         match env::promise_result(0) {
576             PromiseResult::NotReady => unreachable!(),
577             PromiseResult::Successful(value) => {
578                 let new_total_staked_balance = near_sdk::serde_json::from_slice::<U128>(&value)
579                     .expect("Failed to deserialize in increase_stake_callback by value.")
580                     .0;
581
582
583                 let mut staker = self.internal_get_staker_or_panic(&staker_id);
584
585
586                 let sequence = U64(self.next_sequence());
587
588
589                 let staker_new_balance =
590                     self.internal_use_staking_pool_or_panic(&pool_id, |staking_pool| {
591                         let increase_shares = staking_pool.stake(
592                             &mut staker,
593                             stake_amount.0,
594                             new_total_staked_balance,
595                         );
596                         staking_pool.unlock();
597
598
599                         Event::StakerStake {
600                             staking_pool_info: &staking_pool.into(),
601                             staker_info: &(&staker).into(),
602                             select_pool: &staking_pool.pool_id,
603                             stake_amount: &stake_amount,
604                             increase_shares: &increase_shares.into(),
605                             sequence: &sequence,
606                         }
607                         .emit();
608                         staking_pool.staked_amount_from_shares_balance_rounded_down(staker.shares)
609                     });
610                 self.internal_save_staker(&staker_id, &staker);
611
612
613                 return PromiseOrValue::Value(Some(StakingChangeResult {
614                     sequence,
615                     new_total_staked_balance: staker_new_balance.into(),
616                     withdrawal_certificate: None,
617                 }));
618             }
619             PromiseResult::Failed => {
```

```
620                 let mut staker = self.internal_get_staker_or_panic(&staker_id);
621                 staker.select_staking_pool = None;
622                 self.internal_use_staker_or_panic(&staker_id, |staker| {
623                     staker.select_staking_pool = None
624                 });
625                 self.internal_use_staker_staking_pool_or_panic(&staker_id, |pool| pool.unlock());
626                 self.transfer_near(staker_id, stake_amount.0);
627                 emit_callback_failed_event();
628                 return PromiseOrValue::Value(None);
629             }
630         };
631     }
```

Listing 2.82: staking_impl.rs

Impact When a stake operation fails, the users' `select_staking_pool` will not be set to `None`, the `stake pool` will not be unlocked, and the `NEAR` tokens used for staking will not be returned to the user.

Suggestion Unlock the corresponding `stake pool` before setting the user's `select_staking_pool` to `None`.

2.1.31 Potential DoS in Function destroy()

Severity High

Status Confirmed

Introduced by `Version 1`

Description In the function `destroy()` of the contract `LposMarket`, there is a check to ensure the `validator` must be `destroyable`(line 235). One of the conditions is that the `validator's restaking_rewards` should be empty. However, when a `validator` is in the `ToBeDestroyed` state, users are still able to distribute rewards to the `validator` via the functions `ft_on_transfer()` and `distribute_latest_reward()`, which makes `restaking_rewards` unempty. This will prevent the validator from being destroyed.

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

```
230     fn destroy(&mut self, validator_id: AccountId) {
231         self.assert_contract_is_running();
232
233
234         let validator = self.internal_get_validator_or_panic(&validator_id);
235         assert!(validator.is_destroyable());
236
237
238         let withdraw_near = validator.unstake_withdrawn_amount;
239         self.validators.remove(&validator_id);
240         self.escrow_validators.remove(&validator_id);
241
242
243         self.transfer_near(validator_id.clone(), withdraw_near);
244
245
246         Event::Destroy {
247             validator_id: &validator_id,
```

```
            withdraw_near: &withdraw_near.into(),
        }
        .emit();


        ext_escrow::ext(validator.escrow_id)
            .with_attached_deposit(ONE_YOCTO)
            .delete_account(validator_id);
    }
```

Listing 2.83: staking_impl.rs

```
    pub fn is_destroyable(&self) -> bool {
        return matches!(self.status, ValidatorStatus::ToBeDestroyed)
            && self.delegators.is_empty()
            && self.delegators_shares_in_sequence.is_empty()
            && self.delegator_set_in_sequence.is_empty()
            && self.validator_states_in_sequence.is_empty()
            && self.restaking_rewards.is_empty()
            && self.rewards.is_empty();
    }
```

Listing 2.84: validator.rs

```
    fn clean_validator_state(
        &mut self,
        validator_id: AccountId,
    ) -> MultiTxsOperationProcessingResult {
        self.assert_contract_is_running();
        let mut validator = self.internal_get_validator_or_panic(&validator_id);


        assert!(validator.is_able_clean_state());


        let gas_limit = Gas::ONE_TERA.mul(20);
        let delegators = validator.delegators_shares_in_sequence.keys().collect_vec();
        for delegator in delegators {
            if remaining_gas() < gas_limit {
                return MultiTxsOperationProcessingResult::NeedMoreGas;
            }
            let mut shares_in_sequence = validator
                .delegators_shares_in_sequence
                .get(&delegator)
                .unwrap();
            shares_in_sequence.clear();
            validator.delegators_shares_in_sequence.remove(&delegator);
        }


        if remaining_gas() < gas_limit {
            self.internal_save_validator(&validator);
            return MultiTxsOperationProcessingResult::NeedMoreGas;
```

```
219         }
220
221
222         validator.delegator_set_in_sequence.clear();
223
224
225         if remaining_gas() < gas_limit {
226             self.internal_save_validator(&validator);
227             return MultiTxsOperationProcessingResult::NeedMoreGas;
228         }
229         validator.validator_states_in_sequence.clear();
230         self.internal_save_validator(&validator);
231
232
233         return MultiTxsOperationProcessingResult::Ok;
234     }
235 \
```

Listing 2.85: staking_impl.rs

```
80      pub fn is_able_clean_state(&self) -> bool {
81          return matches!(self.status, ValidatorStatus::ToBeDestroyed)
82              && self.restaking_rewards.is_empty();
83      }
```

Listing 2.86: near_ft_impl.rs

```
 8      fn ft_on_transfer(
 9          &mut self,
10          sender_id: AccountId,
11          amount: U128,
12          msg: String,
13      ) -> PromiseOrValue<U128> {
14          let deposit_reward_msg =
15              near_sdk::serde_json::from_str::<FtTransferMessage>(&msg).expect("Invalid msg");
16
17
18          match deposit_reward_msg {
19              FtTransferMessage::AnchorDepositRewardMsg(msg) => {
20                  self.handle_anchor_deposit_reward_msg(sender_id, amount.0, msg)
21              }
22          }
23      }
```

Listing 2.87: validator.rs

```
69      fn distribute_latest_reward(&mut self) {
70          self.assert_contract_is_running();
71          let reward_info = self.anchor_deposit_rewards.pop().expect("No reward found.");
72          log!("Reward information: {:?}", reward_info);
73
74
75          let total_stake_amount: u128 = reward_info.validator_set.iter().map(|(_, b)| b.0).sum();
```

```
76
77
78          assert!(total_stake_amount > 0);
79
80
81          let octopus_commission_amount = (U256::from(reward_info.reward_amount)
82              * U256::from(self.settings.octopus_commission_rate)
83              / U256::from(100))
84          .as_u128();
85          let remain_reward_amount = reward_info
86              .reward_amount
87              .checked_sub(octopus_commission_amount)
88              .unwrap();
89
90
91          self.internal_deposit_octopus_commission(
92              &reward_info.reward_token_id,
93              octopus_commission_amount,
94              &reward_info.reward_uuid,
95          );
96
97
98          let max_validator_staked_amount = reward_info
99              .validator_set
100             .iter()
101             .map(|e| e.1 .0)
102             .max()
103             .expect("Failed to get max_validator_staked_amount.");
104         for (escrow_id, stake_amount) in reward_info.validator_set {
105             let validator_id = self.escrow_validators.get(&escrow_id).unwrap();
106             let validator_receive_reward_amount = (U256::from(stake_amount.0)
107                 * U256::from(remain_reward_amount)
108                 / U256::from(total_stake_amount))
109             .as_u128();
110
111
112             let validator_commission_rate = calculate_validator_commission_rate(
113                 max_validator_staked_amount,
114                 stake_amount.0,
115                 self.settings.validator_commission_rate,
116             );
117
118
119             Event::ValidatorReceiveAnchorReward {
120                 consumer_chain_id: &reward_info.consumer_chain_id,
121                 anchor_id: &reward_info.anchor_id,
122                 reward_token_id: &reward_info.reward_token_id,
123                 reward_token_amount: &validator_receive_reward_amount.into(),
124                 validator_id: &validator_id,
125                 sequence: &reward_info.sequence,
126                 reward_uuid: &reward_info.reward_uuid,
127                 validator_commission_rate: &validator_commission_rate,
128             }
```

```
129             .emit();
130             self.internal_use_validator_or_panic(&validator_id, |validator| {
131                 validator.restaking_rewards.push(&RewardInfo {
132                     reward_token_id: reward_info.reward_token_id.clone(),
133                     reward_amount: validator_receive_reward_amount,
134                     sequence: reward_info.sequence.clone(),
135                     claimed_delegators: UnorderedSet::new(
136                         StorageKey::RewardClaimedDelegatorInValidator {
137                             validator_id: validator.validator_id.clone(),
138                             timestamp: env::block_timestamp(),
139                         },
140                     ),
141                     reward_uuid: reward_info.reward_uuid.clone(),
142                     validator_commission_rate: validator_commission_rate,
143                 })
144             });
145         }
146     }
```

Listing 2.88: restaking_impl.rs

Impact The validator cannot be destroyed even when it is in the `ToBeDestroyed` state..

Suggestion Add checks to ensure that new data cannot be added to `self.restaking_rewards` when the validator is in the `ToBeDestroyed` state.

Feedback from the Project The team may decide to upgrade contract `LposMarket's` when this issue really happens.

2.1.32 Potential Panic in Function transfer_near()

Severity High

Status Confirmed

Introduced by `Version 1`

Description In the function `transfer_near()`, if the `NEAR` amount in the transfer is 0 or if the contract account has an insufficient balance, the function will panic, leading to the failure of `NEAR` transfer. Since this function is used in multiple callback functions, the panic should not occur, or the status rollbacks will fail as well.

```
110     pub(crate) fn transfer_near(&self, account_id: AccountId, amount: Balance) {
111         assert!(amount > 0, "Failed to send near because the amount is 0.");
112         assert!(account_available_amount() >= amount, "Failed to send near because account
                available balance less than the amount.");
113         log!("transfer {} to {}", amount, account_id);
114         Promise::new(account_id).transfer(amount);
115     }
```

Listing 2.89: lib.rs

Impact If the panic in handling the failed promise result occurs within a callback function, both the status rollbacks and the operation to return `NEAR` tokens to the user will fail.

Suggestion Avoid causing a panic in the function `transfer_near()`, provide the team with the proper error log for refunding.

Feedback from the Project The team promises to monitor all failed actions associated with this contract.

2.2 Additional Recommendation

2.2.1 Redundant code

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In contract `RestakingBaseContract`, the check within the function `update_consumer_chain_info()` to ensure the caller is the privileged `governance` account can be completely replaced by the function `assert_cc_gov()` in `consumer_chain.rs`.

```
112    fn update_consumer_chain_info(
113        &mut self,
114        consumer_chain_id: ConsumerChainId,
115        update_param: ConsumerChainUpdateParam,
116    ) {
117        self.assert_contract_is_running();
118        assert_one_yocto();
119        let mut consumer_chain = self.consumer_chains.get(&consumer_chain_id).expect(
120            format!(
121                "ConsumerChain not exist when update_consumer_chain_info by this chain_id: {}",
122                consumer_chain_id
123            )
124            .as_str(),
125        );
126        // check if predecessor is consumer chain governance
127        assert_eq!(
128            consumer_chain.governance,
129            env::predecessor_account_id(),
130            "Only cc_gov({}) can update_consumer_chain_info",
131            consumer_chain.governance
132        );
133
134
135        // Update unbonding period for every stakers.
136        if let Some(new_unbonding_period) = update_param.unbonding_period {
137            if new_unbonding_period != consumer_chain.unbonding_period {
138                for staker_id in consumer_chain.bonding_stakers.iter() {
139                    self.internal_use_staker_or_panic(&staker_id, |staker| {
140                        staker.update_unbonding_period(&consumer_chain_id, new_unbonding_period)
141                    });
142                }
143            }
144        }
145
146
147        consumer_chain.update(update_param.clone());
```

```
148
149
150         self.consumer_chains
151             .insert(&consumer_chain_id, &consumer_chain);
152
153
154         Event::UpdateConsumerChain {
155             consumer_chain_info: &consumer_chain.into(),
156             consumer_chain_update_param: &update_param,
157         }
158         .emit();
159     }
```

Listing 2.90: restaking_impl.rs

```
32     pub fn assert_cc_gov(&self) {
33         let predecessor_account_id = env::predecessor_account_id();
34
35         assert_eq!(
36             predecessor_account_id, self.governance,
37             "The predecessor_account_id({}) is not consumer chain governance({})",
38             predecessor_account_id, self.governance
39         );
40     }
```

Listing 2.91: consumer_chain.rs

Suggestion I Replace above implementation with the function `assert_cc_gov()`.

2.2.2 Lack of Validation for Register Fee

Status Confirmed

Introduced by `Version 1`

Description In `owner_impl.rs`, the `owner` is allowed to set the fee amount for users to register a chain through the function `set_cc_register_fee()`. However, there is no limitation for the maximum of the `new_cc_register_fee`.

The above issue also occurs in the function `set_deploy_fee()` in `owner_impls.rs`.

```
112     fn set_cc_register_fee(&mut self, new_cc_register_fee: U128) {
113         assert_one_yocto();
114         self.assert_owner();
115         self.cc_register_fee = new_cc_register_fee.into();
116     }
```

Listing 2.92: Owner_impl.rs

```
6     fn set_deploy_fee(&mut self, new_deploy_fee: U128) {
7         assert_one_yocto();
8         self.assert_owner();
9         self.settings.validator_deploy_fee = new_deploy_fee.0
10    }
```

Listing 2.93: Owner_impl.rs

Suggestion I Add a check to ensure the fee has a maximum value.

2.2.3 Lack of Check in Function delegate

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `delegate()`, users can delegate their `NEAR` to validators. However, this process lacks sufficient checks, enabling validators to delegate to themselves. This is meaningless.

```
253     #[payable]
254     fn delegate(&mut self, validator_id: AccountId, delegate_amount: U128) {
255         self.assert_contract_is_running();
256         self.assert_max_gas();
257
258
259         assert!(delegate_amount.0 > 0);
260         assert_eq!(STORAGE_FEE + delegate_amount.0, env::attached_deposit());
261
262
263         let delegator_id = env::predecessor_account_id();
264         let delegator = self.internal_get_delegator_or_insert_new(&delegator_id);
265
266
267         assert!(
268             delegator.share_balance == 0 && delegator.select_validator_id.is_none(),
269             "Failed to delegate, already stake in {:?}.",
270             delegator.select_validator_id
271         );
272
273
274         assert!(
275             delegate_amount.0 >= self.settings.minimum_delegator_stake_amount,
276             "Failed to stake, attach near({}) less than minimum_delegator_stake_amount.({}) ",
277             env::attached_deposit(),
278             self.settings.minimum_delegator_stake_amount
279         );
280         let validator = self.internal_get_validator_or_panic(&validator_id);
281
282
283         assert!((validator.delegators.len() as u32) <= self.settings.maximum_delegators_limit);
284         assert!(matches!(validator.status, ValidatorStatus::Staking));
285
286
287         self.ping(validator.validator_id).then(
288             Self::ext(env::current_account_id())
289                 .with_static_gas(Gas::ONE_TERA.mul(TGAS_FOR_INCREASE_STAKE_AFTER_PING))
290                 .delegate_after_ping(delegator_id, validator_id, delegate_amount),
291         );
292     }
```

Listing 2.94: staking_impl.rs

Suggestion I Add checks to prevent validators from delegating themselves.

2.2.4 Incorrect Error Message

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In function `calculate_increase_shares()`, the error message `Increase delegation amount should be positvie` should be `Increase near amount should be positive`.

```
48    pub fn calculate_increase_shares(&self, increase_near_amount: Balance) -> ShareBalance {
49        assert!(
50            increase_near_amount > 0,
51            "Increase delegation amount should be positvie"
52        );
53        let increase_shares =
54            self.share_balance_from_staked_amount_rounded_down(increase_near_amount);
55        assert!(
56            increase_shares>0,
57            "Invariant violation. The calculated number of stake shares for unstaking should be
                  positive"
58        );
59
60
61        let charge_amount = self.staked_amount_from_shares_balance_rounded_down(increase_shares);
62        assert!(
63            charge_amount > 0 && increase_near_amount >= charge_amount,
64            "charge_amount: {}, increase_near_amount: {}",
65            charge_amount,
66            increase_near_amount
67        );
68        increase_shares
69    }
```

Listing 2.95: staking_pool.rs

Suggestion I Replace `Increase delegation amount should be positvie` with `Increase near amount should be positive`.

2.2.5 Refunding Excessive Registration Fee to Incorrect Recipients

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In `storage_management_impl.rs`, users have to register an account for further operations. They can choose to register either for themselves or for others by depositing a specific amount of `storage_fee` through the function `storage_deposit()`. If the specific `account_id` is already registered, the contract will refund `NEAR` to the address corresponding to this `account_id` instead of the sponsor (i.e., `env::predecessor_account_id()`). Similarly, if the account is already registered but the deposited `NEAR` exceeds the `REGISTER_STORAGE_FEE`, the excess part will also be refunded to that address. This design is unreasonable. The above issue also occurs in the function `set_deploy_fee()` in `owner_impls.rs`.

```
11    fn storage_deposit(
12        &mut self,
```

```
13      account_id: Option<AccountId>,
14      #[allow(unused)] registration_only: Option<bool>,
15  ) -> StorageBalance {
16      let account_id = account_id.unwrap_or(env::predecessor_account_id());
17      let exist = self.accounts.contains_key(&account_id);
18      if exist {
19          self.transfer_near(account_id.clone(), env::attached_deposit())
20      } else {
21          assert!(env::attached_deposit() >= REGISTER_STORAGE_FEE);
22          self.internal_save_account(&account_id, &Account::new(account_id.clone()));
23          if env::attached_deposit() > REGISTER_STORAGE_FEE {
24              self.transfer_near(
25                  account_id.clone(),
26                  env::attached_deposit() - REGISTER_STORAGE_FEE,
27              )
28          }
29      }
30
31
32      self.storage_balance_of(account_id).unwrap()
33  }
```

Listing 2.96: storage_management_impl.rs

Suggestion I Refund NEAR to sponsors.

2.2.6 Lack of Check on Account's NEAR Balance

Status Fixed in Version 2

Introduced by Version 1

Description The function transfer_near() of contract LposMarket is designed to send NEAR to other accounts. However, this function lacks a check on the current contract account's NEAR balance. If the amount of NEAR sent exceeds the current account's balance, the cross-contract execution will fail.

```
121   pub(crate) fn transfer_near(&self, account_id: AccountId, amount: Balance) {
122       assert!(amount > 0, "Failed to send near because the amount is 0.");
123       Promise::new(account_id).transfer(amount);
124   }
```

Listing 2.97: lib.rs

Suggestion I Add a check to ensure the contract account has sufficient NEAR balance before transfer.

2.3 Notes

2.3.1 Potential Centralization Problem

Introduced by version 1

Description The protocol has potential centralization risks. In the current implementation, validators establish a binding relationship with the selected consumer chain through the bond operation to receive

rewards distributed by the `consumer chain`. The privileged account, `pos_account`, of the `consumer chain` has the ability to perform `slash` operations on the bonded `validators`, penalizing them for any illegal actions. However, if this privileged account is compromised or its information is leaked, the `validator's` stake in the pool can be drained through such operations.