Security Audit Report for NearOinDao

Date: Dec 4th, 2021

Version: 1.0

Contact: contact@blocksecteam.com

Contents

Report Manifest

Item	Description
Client	Oinfinance
Target	NearOinDao

Version History

Version	Date	Description
1.0	Dec 04, 2021	First Release

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

Chapter 1 Introduction

1.1 About Target Contracts

The target contracts contain a stable coin module. Around it, it also implements other modules, including Staking and Farming. These modules create a positive feedback loop for the stabilization of the stable coin, i.e., USDO.

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

The repositories that have been audited include NearOinDao [1]

The auditing process is iterative. Specifically, we will further audit the commits that fix the founding issues. If there are new issues, we will continue this process. Thus, there are multiple commit SHA values referred in this report. The commit SHA values before and after the audit are shown in the following.

Before and during the audit

Project		Commit SHA	Commit Time
NearOinDao	Commit-1	45d687ecd6b0a0b7d0dc15364f60323650477891	2021.10.27
	Commit-2	d578130518388b5e37d2c84908c571db02182bce	2021.11.01
	Commit-3	cf19bdc5024a95ae415bb67f74129ce7fde6fc4b	2021.11.10
	Commit-4	f9f8691c82857bfddb7e0e39d30003282805e4df	2021.11.27

After

Project	Commit SHA
NearOinDao	3bd117606c753d3c2f66b6dcddd1ae18ea47a20a

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 do 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.

[1] https://github.com/oinfinance/NearOinDao2.1

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 Rust language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

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

We show the main concrete checkpoints in the following.

1.3.1 Software Security

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

1.3.2 DeFi Security

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

1.3.3 NFT Security

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

1.3.4 Additional Recommendation

- Gas optimization
- Code quality and style

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

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. Accordingly, the severity measured in this report are classified into four categories: **High**, **Medium**, **Low** and **Undetermined**.

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

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

Chapter 2 Findings

In total, we find 22 potential issues in the smart contract. We also have 12 recommendation, as follows:

- High Risk: 19
- Medium Risk: 2
- Low Risk: 1
- Recommendations: 12

The details are provided in the following sections.

ID	Severity	Description	Category	Status
1	High	*Logic error while* `self.liquidation_line` *is modified*	Software Security	Confirmed and fixed
2	High	*Function* `liquidation` *may not work*	Software Security	Confirmed and fixed
3	High	*Logic error while setting the time stamp for opening the contract*	Software Security	Confirmed and fixed
4	High	*Contract state is not reverted if the cross contract transaction is failed*	Software Security	Confirmed and fixed
5	High	*Anyone can add the balance of reward*	DeFi Security	Confirmed and fixed
6	High	*Anyone can add the balance of stable pool reward*	DeFi Security	Confirmed and fixed
7	High	*Anyone can burn the other users' coins*	DeFi Security	Confirmed and fixed
8	High	*Anyone can add the balance of their account*	DeFi Security	Confirmed and fixed
9	High	*Oracle does not check the time interval*	DeFi Security	Confirmed and fixed
10	High	*Oracle time interval is too long*	DeFi Security	Confirmed and fixed
11	High	*No oracle for Oin price*	DeFi Security	Confirmed and fixed
12	High	*Users can gain extra reward*	DeFi Security	Confirmed and fixed
13	High	*Users can pay less stable fee*	DeFi Security	Confirmed and fixed
14	Middle	*The multi-signed request can be confirmed with a relatively low confirmation ratio*	DeFi Security	Confirmed and fixed
15	Middle	*Block number per year is inaccurate*	DeFi Security	Confirmed and fixed
16	High	*Available minted coins is not right*	DeFi Security	Confirmed and fixed
17	High	*Payment of stable fee can result in the loss of user's deposited tokens*	DeFi Security	Confirmed and fixed
18	High	*Incorrect staking ratio*	DeFi Security	Confirmed and fixed
19	Low	*Reward coins can beyond the limitation*	DeFi Security	Confirmed and fixed
20	High	*Same whitelist for users in different priveledges*	DeFi Security	Confirmed and fixed
21	High	*No check on the address of stable fee*	DeFi Security	Confirmed and fixed
22	High	*Reward coin's total_reward can be modified by multi-Signature managers*	DeFi Security	Confirmed and fixed
23	-	*Redundant assertion*	Recommendation	Confirmed and fixed

24	-	*Repeated consideration of the liquidation line*	Recommendation	Confirmed and fixed
25	-	*Redundant whitelist check*	Recommendation	Confirmed and fixed
26	-	*Unused function*	Recommendation	Confirmed and fixed
27	-	*Redundant Code*	Recommendation	Confirmed and fixed
28	-	*The function name and the implementation is conflict*	Recommendation	Confirmed and fixed
29	-	*Redundant Code*	Recommendation	Confirmed and fixed
30	-	*The calculation precision can be enhanced*	Recommendation	Confirmed and fixed
31	-	*System may not record previously poked price*	Recommendation	Confirmed and fixed
32	-	*Discontinuous distribution of collateral token in liquidation*	Recommendation	Confirmed and fixed
33	-	*Optimization of calculation precision is not necessary*	Recommendation	Confirmed and fixed
34	-	*The risk of centralized design*	Recommendation	Acknowledged

2.1 Software Security

2.1.1 Two different attributes for the same usage

Status Confirmed and fixed

Description This issue is introduced in or before Commit-1. Two attributes (i.e., `self.cost` and `self.liquidation_line`) represent the same contract state, which is the user's liquidation line. They are used in different functions of the contract (Listing 2.1 and Listing 2.2). However, `self.liquidation_line` can be modified with the function `set_liquidation_line` while `self.cost` cannot be changed. In this case, if the `self.liquidation_line` is modified, `self.cost` keeps the original value. This can influence the logic of the function `assert_user_ratio` (Listing 2.1).

```
530    pub(crate) fn assert_user_ratio(&self) {
531        let user_ratio = self.internal_user_ratio(env::predecessor_account_id());
532        if user_ratio != 0 {
533            assert!(user_ratio >= self.cost, "User ratio less than standard.");
534        }
535    }
```

Listing 2.1: assert_user_ratio:lib.rs

```
585    // TODO liquidation
586    #[payable]
587    pub fn liquidation(&mut self, account: AccountId) {
588        assert!(self.is_liquidation_paused(), "{}", SYSTEM_PAUSE);
589        let ratio = self.internal_user_ratio(account.clone());
590        assert!(ratio > 0, "No current pledge");
591        assert!(ratio <= self.liquidation_line, "Not at the clearing line");
592        ...
```

Listing 2.2: internal_can_mint_amount:lib.rs

Impact The users' liquidation line is not consistent in the different functions of the contract, which influences the logic of the whole contract.

Suggestion I We can unify the usages of these two attributes when calculating the user's staking ratio and comparing it to the system's liquidation line.

2.1.2 Invalid distribution of the liquidation reward

Status Confirmed and fixed

Description This issue is introduced in or before Commit-4. The liquidation sender's account and the contract owner's account may not be registered (Line 193 and 206 of List 2.3). In this case, when the sender aims to conduct liquidation action, the transaction can not be executed successfully due to the raised exception that accounts are not registered.

```
176    pub(crate) fn personal_liquidation_token(&mut self, send_id: AccountId, account_id: AccountId,
           liquidation_gas: Balance, surplus_token: Balance, liquidation_fee: Balance) {
177        //self.owner_id
178        let coin_id = ST_NEAR.to_string();
179        let mut sys_reward_coin = self.internal_get_reward_coin(coin_id.clone());
```

```
180
181         let account_reward_key_o = self.get_staker_reward_key(send_id.clone(), coin_id.clone());
182         let user_reward_coin_o = self.internal_get_account_reward(send_id.clone(), coin_id.clone())
                ;
183
184         self.account_reward.insert(
185             &account_reward_key_o,
186             &UserReward {
187                 index: user_reward_coin_o.index,
188                 reward: user_reward_coin_o.reward.checked_add(liquidation_gas).expect(ERR_ADD),
189             },
190         );
191
192         let account_reward_key_t = self.get_staker_reward_key(account_id.clone(), coin_id.clone());
193         let user_reward_coin_t = self.internal_get_account_reward(account_id.clone(), coin_id.clone
                ());
194
195         if surplus_token > 0 {
196             self.account_reward.insert(
197                 &account_reward_key_t,
198                 &UserReward {
199                     index: user_reward_coin_t.index,
200                     reward: user_reward_coin_t.reward.checked_add(surplus_token).expect(ERR_ADD),
201                 },
202             );
203         }
204
205         let account_reward_key_s = self.get_staker_reward_key(self.owner_id.clone(), coin_id.clone
                ());
206         let user_reward_coin_s = self.internal_get_account_reward(self.owner_id.clone(), coin_id.
                clone());
207
208         self.account_reward.insert(
209             &account_reward_key_s,
210             &UserReward {
211                 index: user_reward_coin_s.index,
212                 reward: user_reward_coin_s.reward.checked_add(liquidation_fee).expect(ERR_ADD),
213             },
214         );
215
216         sys_reward_coin.total_reward = sys_reward_coin
217             .total_reward
218             .checked_add(liquidation_gas).expect(ERR_ADD)
219             .checked_add(liquidation_fee).expect(ERR_ADD)
220             .checked_add(surplus_token).expect(ERR_ADD);
221
222         self.reward_coins.insert(&coin_id, &sys_reward_coin);
223     }
224
225 }
```

Listing 2.3: personal_liquidation_token:reward.rs

Impact Function `liquidation` cannot be executed successfully due to the raised exception that the accounts are not registered.

Suggestion I Assert the existence of the liquidation sender's account and the contract owner's account at the beginning of function `liquidation` .

2.1.3 Block_timestamp is saved to the closed_time while opening the system

Status Confirmed and fixed

Description This issue is introduced in or before Commit-3. `env::block_time_stamp()` should not be saved to the `self.closed_time` when invoking the function `internal_open`.

```
109    #[private]
110    pub fn internal_open(&mut self) {
111        self.closed_time = env::block_timestamp();
112        self.open_stake();
113        self.open_redeem();
114        self.open_claim_reward();
115        self.open_liquidation();
116        self.open_stable();
117        log!(
118            "{} open sys in {}",
119            env::predecessor_account_id(),
120            self.closed_time
121        );
122    }
```

Listing 2.4: internal_open:esm.rs

Impact The opening time and closed time of the contract is completely wrong. Further updates that depend on the time information can have logic error.

Suggestion I We suggest to create a new contract state called `self.opening_time` and assigned the `env::block_timestamp()` to this value while invoking opening the contract.

2.1.4 Contract state is not reverted when the cross function calls are failed

Status Confirmed and fixed

Description This issue is introduced in or before Commit-3. The process of storage_deposit and ft_transfer may fail during the cross contract function calls. We cannot guarantee that the transfer will always be performed correctly. The callback function does not revert the contract state if the call is failed.

```
160    #[private]
161    pub fn storage_deposit_callback(&mut self) {
162        match env::promise_result(0) {
163            PromiseResult::NotReady => unreachable!(),
164            PromiseResult::Successful(_) => {
165                log!("Transfer success");
166            }
167            PromiseResult::Failed => {
168                log!("Transfer failed");
169            }
```

```
170         }
171     }
```

Listing 2.5: storage_deposit_callback:ft.rs

```
173     #[private]
174     pub fn liquidation_transfer_callback(&mut self) {
175         match env::promise_result(0) {
176             PromiseResult::NotReady => unreachable!(),
177             PromiseResult::Successful(_) => {
178                 log!("Transfer success");
179             }
180             PromiseResult::Failed => {
181                 log!("Transfer failed");
182             }
183         }
184     }
```

Listing 2.6: liquidation_transfer_callback:ft.rs

Impact Users may loss their assets when transactions failed as the callback function does not revert the contract state.

Suggestion I We need to revert the contract state (when the transfer fails) in the callback function of the cross contract function calls.

2.2 DeFi Security

2.2.1 inject_reward lacks access control

Status Confirmed and fixed.

Description This issue is introduced in or before Commit-1. Function `inject_reward` is public. Anyone can invoke this function to add the balance of the reward in the contract.

```
33     pub fn inject_reward(&mut self, amount: U128, reward_coin: AccountId) {
34         // self.assert_owner();
35
36         if reward_coin == String::from("NEAR") {
37             assert!(
38                 amount.0 == env::attached_deposit(),
39                 "Amount not equal transfer_amount"
40             );
41         }
42         ...
43     }
```

Listing 2.7: inject_reward:pool.rs

Impact Anyone can add arbitrary balance on the reward of the contract.

Suggestion I This function should be changed as a private one as it is called internally after receiving the transferred reward.

2.2.2 inject_sp_reward lacks access control

Status Confirmed and fixed.

Description This issue is introduced in or before Commit-1. Function `inject_sp_reward` is public. Anyone can invoke this function to add the balance of the stable pool reward in the contract.

```
1092    pub fn inject_sp_reward(&mut self, _amount: U128, sender_id: ValidAccountId) {
1093        self.reward_sp = self.reward_sp + u128::from(_amount);
1094
1095        log!(
1096            "{} add sp_reward {} cur amount{}",
1097            sender_id,
1098            u128::from(_amount),
1099            self.reward_sp
1100        );
1101    }
```

Listing 2.8: inject_sp_reward:stablepool.rs

Impact Anyone can add arbitrary balance on the stable pool reward of the contract.

Suggestion I This function should be changed as a private one as it is called internally after receiving the transferred stable pool reward.

2.2.3 burn_coin lacks access control

Status Confirmed and fixed.

Description This issue is introduced in or before Commit-1. Function `burn_coin` is public. Anyone can invoke this function to burn anyone's coin.

```
742    pub fn burn_coin(&mut self, amount: U128, fee: Balance, sender_id: ValidAccountId) -> Balance{
743        assert!(self.is_redeem_paused(), "{}", SYSTEM_PAUSE);
744        let sender_id = AccountId::from(sender_id);
745        self.assert_is_poked();
746        self.accured_token(sender_id.clone());
747        ...
748    }
```

Listing 2.9: burn_coin:lib.rs

Impact Anyone can use this function to burn anyone's coin, resulting the loss of users' assets.

Suggestion I This function should be changed as a private one as it is called internally after receiving the transferred stable fee for burning coins.

2.2.4 deposit_token lacks access control

Status Confirmed and fixed.

Description This issue is introduced in or before Commit-1. Function `deposit_token` is public. Anyone can invoke this function to add the balance of their account.

```
377    pub fn deposit_token(&mut self, amount: u128, _sender_id: ValidAccountId) {
378        self.assert_is_poked();
379        assert!(self.is_stake_paused(), "{}", SYSTEM_PAUSE);
380        let _amount = u128::from(amount);
381        let sender_id = AccountId::from(_sender_id);
382        . . .
383    }
```

Listing 2.10: deposit_token:lib.rs

Impact Attackers can invoke this function to add the balance of their account.

Suggestion I This function should be changed as a private one as it is called internally after receiving the deposited tokens.

2.2.5 Oracle lacks the check of time

Status Confirmed and fixed.

Description This issue is introduced in or before Commit-1. The function `assert_is_poked` in `oracle.rs` only checks whether the value of the token price is zero. This does not makes sense as the token price is keep changing.

```
55    pub(crate) fn assert_is_poked(&self) {
56        assert!(self.token_price != 0, "Oracle price isn't poked.");
57    }
```

Listing 2.11: assert_is_poked:oracle.rs

Impact This issue affects price oracles. If the token price hasn't been poked for a quiet long time, the assert can still be passed and related transaction can be executed with an outdated price.

Suggestion I The contract should set a valid time period for the poked price.

2.2.6 Inappropriate oracle poke interval time

Status Confirmed and fixed.

Description This issue is introduced in or before Commit-3. The constant `POKE_INTERVAL_TIME` defined in types.rs means 1000 days now. And this time interval seems too long. A reasonable value is required.

```
69pub const POKE_INTERVAL_TIME: u64 = 86_400_000_000_000_000;
```

Listing 2.12: types.rs

Impact The time interval for poked price is inappropriate.

Suggestion I Reset the interval time for poked price with a reasonable value.

2.2.7 Missing Assert for Oin_Price

Status Confirmed and fixed.

Description This issue is introduced in or before Commit-1. This function does not check whether the value of the oin_token price is poked since user's stable fee is calculated by the `self.oin_price`.

```
624     pub fn internal_user_stable(&self, account: AccountId) -> u128 {
625         let user_stable = self.account_stable.get(&account).expect("error");
626         let allot = self.get_account_allot(account.clone());
627         let coin = self
628             .account_coin
629             .get(&account)
630             .expect("error")
631             .checked_add(allot.0)
632             .expect(ERR_ADD);
633         let current_block_number = env::block_timestamp().checked_div(INIT_BLOCK_TIME).expect(
                ERR_DIV);
634         user_stable
635             .saved_stable
636             .checked_add(
637                 self.stable_fee_rate//16
638                     .checked_div(BLOCK_PER_YEAR)
639                     .expect(ERR_DIV)
640                     .checked_mul(current_block_number as u128 - user_stable.block)
641                     .expect(ERR_MUL)
642                     .checked_mul(coin)//8
643                     .expect(ERR_MUL)
644                     .checked_div(self.oin_price)//8
645                     .expect(ERR_DIV)
646                     .checked_div(ONE_COIN)//8
647                     .expect(ERR_DIV),
648             )
649             .expect(ERR_ADD)
650     }
```

Listing 2.13: internal_user_stable:lib.rs

Impact The outdated OIN price may lead to price manipulation without checking the freshness of the price poked by the oracle.

Suggestion I Add a `self.assert_is_poked();` assertion before the calculation of user's stabel fee.

2.2.8 Users may gain more mining reward with staking token

Status Confirmed and fixed.

Description This issue is introduced in or before Commit-1. The claimed reward is not calculated accurately. Function `internal_get_saved_reward` is called to calculate the user's specific mining reward from t0 to t1 with the following formula:

$$(reward\_coin\_ins.index - user\_reward.index) * (account.token + account\_allot.token)$$

Note that account_allot.token is the collateral reward added by other user's liquidation. However, liquidation may happen at any time from t0 to t1. For example, a user deposited 100 token on day0. On day999, liquidation for the other user is triggered so that `account_allot.token` may increased to 1000.

When the user claims his reward on day1000, the 1000 token resulted from liquidation on day999 should only be counted for mining for one day. However, the contract actually calculate the mining reward for the collateral reward from day0 to day1000.

```
61  // TODO[OK] Calculation of reward
62  pub(crate) fn internal_get_saved_reward(
63      &self,
64      staker: AccountId,
65      reward_coin: AccountId,
66  ) -> u128 {
67      let reward_coin_ins = self.internal_get_reward_coin(reward_coin.clone());
68      let (stake_token_num, _) = self.staker_debt_of(staker.clone());
69
70      if let Some(user_reward) = self
71          .account_reward
72          .get(&self.get_staker_reward_key(staker.clone(), reward_coin.clone()))
73      {
74          user_reward
75              .reward
76              .checked_add(
77                  U256::from(
78                      reward_coin_ins
79                          .index
80                          .checked_sub(user_reward.index)
81                          .expect(ERR_SUB),
82                  )
83                  .checked_mul(U256::from(stake_token_num))
84                  .expect(ERR_MUL)
85                  .checked_div(U256::from(reward_coin_ins.double_scale))
86                  .expect(ERR_DIV)
87                  .as_u128(),
88              )
89              .expect(ERR_ADD)
90      } else {
91          0
92      }
93  }
```

Listing 2.14: internal_get_saved_reward:views.rs

```
27  pub fn staker_debt_of(&self, staker: AccountId) -> (u128, u128) {
28      if let Some(token) = self.account_token.get(&staker) {
29          let coin = self.account_coin.get(&staker).expect(ERR_NOT_REGISTER);
30          let allot = self.get_account_allot(staker.clone());
31          (token + allot.1, coin + allot.0)
32      } else {
33          (0, 0)
34      }
35  }
```

Listing 2.15: staker_debt_of:views.rs

Impact Users may gain extra rewards.

Suggestion I Remove the partition of newly allocated collateral when calculating mining reward. We can make the mining reward only related to the amount of tokens deposited by the user.

2.2.9 Users may pay less stable fee

Status Confirmed and fixed.

Description This issue is introduced in or before Commit-1. Suppose one user mints 1000 USDOs on day 0, and the `stable_fee_rate` at that time is 0.01oin/coin/day. If the user returns back the 1000 USDOs on day100 and the stable fee rate does not change during the past 100 days, the stable fee he needs to pay is 0.01 Oin/coin/day * 1000 Coin * 100 Day = 1000 Oin. However, if the owner set the `stable_fee_rate` = 0.005 oin/coin/day on day99. In this time, the user only needs to pay 0.005 Oin/Coin/Day * 1000 Coin * 100 Day = 500 Oin. In fact, the accurate fee should be: (0.01 Oin/Coin/Day * 1000 Coin * 99 Day) + (0.005 Oin/Coin/Day * 1000 Coin * 1 Day) = 990 Oin + 5 Oin = 995 Oin.

In this case, the 495 Oin are not required to be paid by users.

```
33    // TODO [OK]
34    pub fn set_stable_fee_rate(&mut self, fee_rate: U128) {
35        self.assert_param_white();
36        self.update_stable_index();
37        assert!(fee_rate.0 <= INIT_MAX_STABLE_FEE_RATE, "Exceeding the maximum setting");
38        self.stable_fee_rate = fee_rate.into();
39        log!("Set stable fee rate {}", fee_rate.0);
40    }
```

Listing 2.16: set_stable_fee_rate:dparam.rs

```
57    pub fn update_stable_index(&mut self) {
58    }
```

Listing 2.17: update_stable_index:stablefee.rs

Impact Contract users may be charged less for stable fee.

Suggestion I Implement the stable fee's system index like the calculation of reward_coin in this contract. And make sure that the stable fee's system index is updated whenever `set_stable_fee_rate`, `liquidation` and `update_stable_fee` is called by contract users.

2.2.10 Unreasonable multi-signed request confirmation rate

Status Confirmed and fixed.

Description This issue is introduced in or before Commit-1. The multi-signed request confirmation rate is calculated by the number of multi-signature managers when the request was created. But the number of multi-signature managers may change later. In this case, if the number of managers increases, the request can be confirmed with a low confirmation ratio.

```
182   pub(crate) fn is_num_enough(&self, request_id: RequestId) -> bool {
183       let request = self.requests.get(&request_id).unwrap();
184       let confirmations = self.confirmations.get(&request_id).unwrap();
185
186       let num_confirmrations = request.num_confirm_ratio * (request.mul_white_num);
187       log!(
188           "confim num is {} num needed is {} ",
189           confirmations.len() as u32 * 100,
190           num_confirmrations
```

```
191         );
192
193         (confirmations.len() as u64) * 100 >= num_confirmrations
194     }
```

Listing 2.18: is_num_enough:multisign.rs

```
72      pub fn add_request_only(&mut self, request: MultiSigRequest) -> RequestId {
73          self.assert_mul_white();
74          ...
75
76          let request_added = MultiSigRequestWithSigner {
77              signer_pk: env::signer_account_pk(),
78              added_timestamp: env::block_timestamp(),
79              confirmed_timestamp: 0,
80              request: request,
81              is_executed: false,
82              cool_down: self.request_cooldown,
83              mul_white_num: self.mul_white_num(),
84              num_confirm_ratio: self.num_confirm_ratio,
85          };
86
87          self.requests.insert(&self.request_nonce, &request_added);
88          ...
89      }
```

Listing 2.19: add_request_only:multisign.rs

Impact Multi-signed requests may be confirmed with a low confirmation rate as the contract only consider the number of managers when the request is created.

Suggestion I Consider using the number of multi-signed users in the current contract state to calculate the multi-signed request confirmation rate.

2.2.11 Incorrect block number per year

Status Confirmed and fixed.

Description This issue is introduced in or before Commit-1. Given that a block is generated every second on the NEAR protocol's mainnet, the generated block number per year should be 31536000 (365days) rather than 31104000 (360days).

```
74      pub const BLOCK_PER_YEAR: u128 = 31104000;
```

Listing 2.20: types.rs

Impact Inaccurate constant for `BLOCK_PER_YEAR` will make the results of calculations using the constant inconsistent with reality.

Suggestion I Change the `BLOCK_PER_YEAR` to be 31536000.

2.2.12 Incorrect calculation of the maximum usdo can mint

Status Confirmed and fixed.

Description This issue is introduced in or before Commit-1. `allot_token.0` represents the allocated debt. While calculating the available mint amount for USDO, the allocated debt should not be counted. Otherwise, a user with very high debt can mint a huge number of USDOs.

```
585    pub(crate) fn internal_can_mint_amount(&self, account: AccountId) -> u128 {
586        self.assert_is_poked();
587        let token = self.account_token.get(&account).expect(ERR_NOT_REGISTER);
588        let guarantee = self.guarantee.get(&account).expect(ERR_NOT_REGISTER);
589        let allot_token = self.get_account_allot(account.clone());
590
591        let max_usdo = (U256::from(token)
592            .checked_add(U256::from(allot_token.1))
593            .expect(ERR_ADD))
594        .checked_mul(U256::from(self.token_price))
595        .expect(ERR_MUL)
596        .checked_div(U256::from(self.liquidation_line))
597        .expect(ERR_DIV)
598        .checked_div(U256::from(INIT_STABLE_INDEX))
599        .expect(ERR_DIV)
600        .checked_add(U256::from(allot_token.0))
601        .expect(ERR_ADD)
602        .checked_sub(U256::from(guarantee))
603        .unwrap_or(U256::from(0))
604        .as_u128();
605
606        ...
607    }
```

Listing 2.21: internal_can_mint_amount:lib.rs

Impact Users can mint additional USDOs when invoking the function `mint_coin`.

Suggestion I The `allot_token.0`, which represents the allocated debt, should not be counted as the available minted USDOs.

2.2.13 Incorrect handling of user's stable fee

Status Confirmed and fixed. (The related logic is removed now)

Description This issue is introduced in or before Commit-1. When users invoke the function `burn_coin`, the stable fee is paid with 'OIN' token rather than 'ST_NEAR'. However, the contract will reduce the balance of the user's staking token, which is not accurate.

```
742    pub(crate) fn burn_coin(&mut self, amount: U128, fee: Balance, sender_id: ValidAccountId) ->
           Balance{
743        ...
744            assert!(usdo >= amount.into(), "Insufficient amount");
745            let token = self.account_token.get(&sender_id.clone()).expect(ERR_NOT_REGISTER);
746            self.internal_burn(sender_id.clone(), amount.into());
747
748            self.total_token = self.total_token.checked_sub(unpaid_fee.into()).expect(ERR_SUB);
749            self.account_token.insert(
750                &sender_id.clone(),
751                &token.checked_sub(unpaid_fee.into()).expect(ERR_SUB),
```

```
752             );
753         ...
754
755     }
```

Listing 2.22: burn_coin:lib.rs

Impact Users' staking token can be reduced due to the incorrect handling of the user's stable fee.

Suggestion I Use the correct token for paying the stable fees.

2.2.14 Incorrect system ratio

Status Confirmed and fixed.

Description This issue is introduced in or before Commit-1. If `total_coin` = 0, the ratio should be $+\infty$. Setting it to 0 is incorrect.

```
432     pub(crate) fn internal_sys_ratio(&self) -> u128 {
433         self.assert_is_poked();
434         let token_usd = U256::from(self.total_token)
435             .checked_mul(U256::from(self.token_price))
436             .expect(ERR_MUL); /* 32 */
437         let total_coin = self.total_coin + self.total_guarantee;
438         if total_coin == 0 {
439             0
440         } else {
441             token_usd
442                 .checked_div(U256::from(STAKE_RATIO_BASE))
443                 .expect(ERR_DIV)
444                 .checked_div(U256::from(total_coin))
445                 .expect(ERR_DIV)
446                 .as_u128()
447         }
448     }
```

Listing 2.23: internal_sys_ratio:lib.rs

Impact The system is likely to shut down due to the incorrect ratio.

Suggestion I Change the if condition `total_coin = 0` to `token_usd = 0`.

2.2.15 The number of reward coin can be larger than the upper bound

Status Confirmed and fixed.

Description This issue is introduced in or before Commit-1. When there are 20 reward coins now, the assert at line 131 of Listing 2.24 can be passed. In this case, one more reward coin can be added and the total number of rewards coins can be larger than the `REWARD_UPPER_BOUND`.

```
129     pub(crate) fn internal_add_reward_coin(&mut self, coin: RewardCoin) {
130         assert!(
131             self.reward_coins.len() <= REWARD_UPPER_BOUND,
132             "The currency slot has been used up, please modify other currency information as
                appropriate",
```

```
133         );
134
135         match self.reward_coins.get(&coin.token) {
136             Some(_) => {
137                 env::panic(b"The current currency has been added, please add a new currency.");
138             }
139             None => {}
140         }
141         self.reward_coins.insert(&coin.token, &coin);
142
143         log!(
144             "{} add the RewardCoin=> {:?}",
145             env::predecessor_account_id(),
146             coin
147         )
148     }
```

Listing 2.24: internal_add_reward_coin:pool.rs

Impact The available added number of reward coins is conflicted with the design of the system.

Suggestion I Change the assert into `self.reward_coins.len() < REWARD_UPPER_BOUND`.

2.2.16 Users in different privileges use the same white list

Status Confirmed and fixed.

Description This issue is introduced in or before Commit-1. Functions `assert_param_white`, `assert_white`, `assert_esm_white`,`assert_oracle_white` are used for different privileges. However, they share the same whitelist.

```
147     pub(crate) fn assert_esm_white(&self) {
148         self.assert_white()
149     }
```

Listing 2.25: assert_esm_white:esm.rs

```
117     pub(crate) fn assert_param_white(&self) {
118         self.assert_white();
119     }
```

Listing 2.26: assert_param_white:dparam.rs

```
50     pub(crate) fn assert_oracle_white(&self) {
51         self.assert_white();
52     }
```

Listing 2.27: assert_oracle_white:oracle.rs

Impact Users in different privilege share the same white list.

Suggestion I Implement different white lists for users with different privileges.

2.2.17 burn_coin does not check the token type

Status Confirmed and fixed.

Description This issue is introduced in or before Commit-1. Functions `burn_coin` does not check the token type. In this case, attackers can transfer arbitrary tokens with specified amount for paying the stable fee.

```
742    pub fn burn_coin(&mut self, amount: U128, fee: Balance, sender_id: ValidAccountId) -> Balance{
743        assert!(self.is_redeem_paused(), "{}", SYSTEM_PAUSE);
744        let sender_id = AccountId::from(sender_id);
```

Listing 2.28: assert_esm_white:esm.rs

Impact Users do not need to pay Oin token. Instead, they can pay the stable fee by transfer arbitrary token with the required amount.

Suggestion I Check the address of the received token.

2.2.18 Reward coin's total_reward can be modified by multi-Signature managers

Status Confirmed and fixed.

Description This issue is introduced in or before Commit-3. Function `inject_reward` is decorated with `#[private]`. Therefore, multi-signature managers can invoke this function through multi-signature requests and add arbitrary amount on the total reward without injecting reward.

```
32     #[payable]
33     #[private]
34     pub fn inject_reward(&mut self, amount: U128, reward_coin: AccountId) {
35         // self.assert_owner();
36
37         if reward_coin == String::from("NEAR") {
38             assert!(
39                 amount.0 == env::attached_deposit(),
40                 "Amount not equal transfer_amount"
41             );
42         }
43
44         if let Some(reward_coin_ins) = self.get_reward_coin(reward_coin.clone()) {
45             let mut reward_coin_ins = reward_coin_ins;
46             reward_coin_ins.total_reward = reward_coin_ins
47                 .total_reward
48                 .checked_add(amount.into())
49                 .expect(ERR_SUB);
50             self.reward_coins.insert(&reward_coin, &reward_coin_ins);
51
52             if reward_coin == String::from("NEAR") {
53
54             } else {
55                 log!("Transfer is not required for post-processing");
56             }
57         } else {
58             env::panic(b"No the reward coin.");
59         }
```

```
60    }
```

Listing 2.29: inject_reward:pool.rs

Suggestion I Remove the decorator `#[private]`, and change the visibility of the function `inject_reward` to be private.

2.3 Additional Recommendation

2.3.1 Redundant assertion

Status Confirmed and fixed.

Description This issue is introduced in or before Commit-2. Function `inject_reward` should only be called by `ft_on_transfer` internally. The address of the reward coin is checked in `ft_on_transfer`. In this case, we do not need to check the name of reward coin at the beginning of the function `inject_reward`.

```
37    #[payable]
38    #[private]
39    pub fn inject_reward(&mut self, amount: U128, reward_coin: AccountId) {
40        // self.assert_owner();
41
42        if reward_coin == String::from("NEAR") {
43            assert!(
44                amount.0 == env::attached_deposit(),
45                "Amount not equal transfer_amount"
46            );
47        }
48
49    ...
50    }
```

Listing 2.30: inject_reward:pool.rs

```
900
901   pub fn ft_on_transfer(
902       &mut self,
903       sender_id: ValidAccountId,
904       amount: U128,
905       msg: String, /* token */
906   ) -> PromiseOrValue<U128> {
907   ...
908           FtOnTransferArgs::InjectReward => {
909               assert_eq!(sender_id.to_string(), self.owner_id, "ERR_NOT_ALLOWED");
910
911               assert!(
912                   self.reward_coins.get(&token_account_id).is_some(),
913                   "Invalid reward coin"
914               );
915
916               self.inject_reward(amount, token_account_id);
917               amount_return = 0;
918           }
```

```
919     ...
920     }
```

Listing 2.31: ft_on_transfer:lib.rs

Suggestion I Remove check on the name of reward coin in `inject_reward`.

2.3.2 Repeated assertion for user's liquidation ratio

Status Confirmed and fixed.

Description This issue is introduced in or before Commit-1. The liquidation line is already taken into consideration in function `internal_avaliable_token`, so there is no need to check whether the user_ratio's reaches the liquidation line later.

```
544     #[payable]
545     pub fn withdraw_token(&mut self, amount: U128) {
546         assert!(self.is_stake_paused(), "{}", SYSTEM_PAUSE);
547         let mut amount = amount.0;
548 
549         let token = self.internal_avaliable_token(env::predecessor_account_id());
550         let debt = self.get_dept(env::predecessor_account_id());
551 
552         log!("token :{} amount: {}", token, amount);
553         assert!(token >= amount, "Insufficient avaliable token.");
554         if debt.0 - debt.2 == 0 {
555             if token - amount < self._min_amount_token() {
556                 amount = token;
557             }
558         } else {
559             self.assert_user_ratio();
560             if token - amount < self._min_amount_token() {
561                 env::panic(b"Please return all coins first");
562             }
563         }
```

Listing 2.32: withdraw_token:lib.rs

Suggestion I Remove the redundant assertion in Line 559 of Listing 2.32.

2.3.3 Redundant whitelist check

Status Confirmed and fixed.

Description This issue is introduced in or before Commit-1. Function `set_reward_speed` invoke the function `assert_param_white` to check the privilege. Meanwhile, the `internal_set_reward_speed`, which is called by `set_reward_speed`, invoke the `assert_white` again. `assert_white` has the same whitelist as the `assert_param_white`.

```
154     pub fn set_reward_speed(&mut self, reward_coin: AccountId, speed: U128) {
155         self.assert_param_white();
156         self.internal_set_reward_speed(reward_coin, speed);
157     }
```

Listing 2.33: set_reward_speed:dparam.rs

```
pub(crate) fn internal_set_reward_speed(&mut self, reward_coin: AccountId, speed: U128) {
    self.assert_white();
    self.update_index();
    . . .
}
```

Listing 2.34: internal_set_reward_speed:pool.rs

Suggestion I Remove `assert_white` inside the function `internal_set_reward_speed`.

2.3.4 Unused function

Status Confirmed and fixed.

Description This issue is introduced in or before Commit-3. Function `on_inject_reward` is not used by any other functions. Thus, it can be removed.

```
#[private]
pub fn on_inject_reward(&mut self, reward_coin: AccountId, amount: U128) {
    match env::promise_result(0) {
        PromiseResult::NotReady => unreachable!(),
        PromiseResult::Successful(_) => {}
        PromiseResult::Failed => {
            let mut reward_coin_ins = self.internal_get_reward_coin(reward_coin.clone());
            reward_coin_ins.total_reward = reward_coin_ins
                .total_reward
                .checked_sub(amount.into())
                .expect(ERR_ADD);
            self.reward_coins.insert(&reward_coin, &reward_coin_ins);
        }
    };
}
```

Listing 2.35: on_inject_reward:pool.rs

Suggestion I Remove the function `on_inject_reward`.

2.3.5 Redundant Code

Status Confirmed and fixed.

Description This issue is introduced in or before Commit-3. Function `account_allot.get()` is used to get the allocated reward and debt. Inside the function `set_account_allot`, the invocation of this function is not required.

```
pub(crate) fn set_account_allot(&mut self,account_id: AccountId){
    //Update [personally assigned debt, personally assigned pledge] to system value
    let (allot_debt, allot_token) = self.get_account_allot(account_id.clone());
    let token = self.account_token.get(&account_id).expect(ERR_NOT_REGISTER);
    let coin = self.account_coin.get(&account_id).expect(ERR_NOT_REGISTER);

```

```
42        self.account_allot.get(&account_id);
43
44        self.account_allot.insert(
45            &account_id,
46            &AccountAllot{
47                account_allot_debt: self.sys_allot_debt,
48                account_allot_token: self.sys_allot_token,
49            }
50        );
51        self.account_coin.insert(&account_id, &coin.checked_add(allot_debt).expect(ERR_ADD));
52        self.account_token.insert(&account_id, &token.checked_add(allot_token).expect(ERR_ADD));
53    }
```

Listing 2.36: set_account_allot:allot.rs

Suggestion I Remove the invocation `account_allot.get()` at line 42.

2.3.6 The function name and the implementation is opposite

Status Confirmed and fixed.

Description This issue is introduced in or before Commit-3. Function `is_stake_paused`, `is_redeem_paused`, `is_claim_reward_paused`,`is_liquidation_paused`,`is_stable_paused` are defined to represent whether the function is paused or not. However, when the specific attribute is live, it returns True.

```
72    // TODO [OK]
73    pub(crate) fn is_stake_paused(&self) -> bool {
74        self.stake_live == 1
75    }
76
77    // TODO [OK]
78    pub(crate) fn is_redeem_paused(&self) -> bool {
79        self.redeem_live == 1
80    }
81
82    // TODO [OK]
83    pub(crate) fn is_claim_reward_paused(&self) -> bool {
84        self.claim_live == 1
85    }
86
87    // TODO [OK]
88    pub(crate) fn is_liquidation_paused(&self) -> bool {
89        self.liquidation_live == 1
90    }
91
92    // TODO [OK]
93    pub(crate) fn is_stable_paused(&self) -> bool {
94        self.stable_live == 1
95    }
```

Listing 2.37: is_{stake|redeem|claim_reward|liquidation|stable}_paused:esm.rs

Suggestion I Change the function name of is_{stake|redeem|claim_reward|liquidation|stable}_paused into is_{stake|redeem|claim_reward|liquidation|stable}_live

2.3.7 Redundant Code

Status Confirmed and fixed.

Description This issue is introduced in or before Commit-3. Function `update_stable_fee` is used to update the required stable fees. Stable fees is not related to the staked tokens. Thus, changing the balance of token for users does not need to update the stable fees.

```
331    pub(crate) fn deposit_token(&mut self, _amount: u128, _sender_id: ValidAccountId) {
332        self.assert_is_poked();
333        assert!(self.is_stake_paused(), "{}", SYSTEM_PAUSE);
334        let sender_id = AccountId::from(_sender_id);
335        assert!(_amount > 0, "Deposit token amount must greater than zero.");
336
337        if let Some(0) = self.guarantee.get(&sender_id) {
338            assert!(
339                _amount >= self._min_amount_token(),
340                "Deposit token amount must greater the minimum deposit token."
341            );
342        }
343        self.update_personal_token(sender_id.clone());
344        self.update_stable_fee(sender_id.clone());
345        self.set_account_allot(sender_id.clone());
346        . . .
347    }
```

Listing 2.38: deposit_token:lib.rs

Suggestion I Remove the invocation `update_stable_fee` at line 344.

2.3.8 The calculation precision can be enhanced

Status Confirmed and fixed.

Description This issue is introduced in or before Commit-3. Function `internal_user_stable` aims to calculate the stable fee. The calculation precision can be enhanced by conducting multiplication before division.

```
15    pub(crate) fn update_stable_fee(&mut self, account: AccountId) {
16        if let Some(mut user_stable) = self.account_stable.get(&account) {
17            let allot = self.get_account_allot(account.clone());
18            let debt = allot.0;
19            let current_block_number = self.to_nano( env::block_timestamp()) as u128;
20
21            let coin = self.account_coin.get(&account).expect(ERR_NOT_REGISTER).checked_add(debt).
                  expect(ERR_ADD);
22            let delta_block = current_block_number.checked_sub(user_stable.block).expect(ERR_SUB);
23            if delta_block > 0 && coin > 0 {
24                let fee = self.stable_fee_rate//16
25                        .checked_mul(delta_block).expect(ERR_MUL)
26                        .checked_mul(coin).expect(ERR_MUL)//8
27                        .checked_div(BLOCK_PER_YEAR).expect(ERR_DIV)
28                        .checked_div(self.oin_price).expect(ERR_DIV)//8
29                        .checked_div(ONE_COIN).expect(ERR_DIV);//8
```

```
30
31              self.saved_stable = self.saved_stable
32                      .checked_add(fee).expect(ERR_ADD);
33
34              user_stable.saved_stable = user_stable.saved_stable
35                      .checked_add(fee).expect(ERR_ADD);
36          }
37
38          user_stable.block = current_block_number;
39          self.account_stable.insert(&account, &user_stable);
40          log!("Current stabilization fee: {:?}",self.account_stable.get(&account));
41      } else {
42          env::panic(b"Not register")
43      }
44  }
```

Listing 2.39: update_stable_fee:stablefee.rs

Suggestion I Conduct the multiplication before division for the calculation from line 25 to line 30.

2.3.9 System may not record previously poked price

Status Confirmed and fixed.

Description This issue is introduced in or before Commit-1. The function is not implemented correctly. System may not record poked price as the number of total tokens deposited in the contract is greater than 0 in most cases.

```
26pub fn poke(&mut self, token_price: U128) {
27    ...
28      if self.total_token > 0 {
29          if self.internal_sys_ratio() <= INIT_MIN_RATIO_LINE {
30              self.internal_shutdown();
31          }
32      }else {
33          log!(
34              "{} poke price {} successfully.",
35              env::predecessor_account_id(),
36              token_price.0
37          );
38       }
39  }
```

Listing 2.40: poke:oracle.rs

Suggestion I Recording the behavior of poking token price should not be influenced by the number of deposited tokens in the contract.

2.3.10 Discontinuous distribution of collateral token in liquidation

Status Confirmed and fixed.

Description This issue is introduced in or before Commit-4. When the user's staking ratio is larger or equal than 108.5%, users have to pay the liquidation_fee, which owns 2% of the `allot_debt`. However, if

the user's staking ratio is less than 108.5%, he/she does not need to pay the liquidation fee. This result in the fact that user with larger staking ratio may allot less staking token to the pool after liquidation.

```
820    #[payable]
821    pub fn liquidation(&mut self, account: AccountId) {
822        ...
823        if ratio >= INIT_NO_LIQUIDATION_FEE_RATE {
824            liquidation_fee = _allot_debt
825                            .checked_mul(self.liquidation_fee_ratio).expect(ERR_MUL)
826                            .checked_mul(STAKE_RATIO_BASE).expect(ERR_MUL)//16
827                            .checked_div(self.token_price).expect(ERR_DIV);
828        }else{
829            allot_ratio = ratio
830                .checked_sub(self.gas_compensation_ratio).expect(ERR_SUB)
831                .checked_add(1).expect(ERR_ADD);
832        }
833        ...
```

Listing 2.41: liquidation:lib.rs

Suggestion I For user whose staking ratio is between 108.5% to 110.5%, the liquidation fee is suggested to be (staking ratio - 108.5%).

2.3.11 Optimization of calculation precision is not necessary

Status Confirmed and fixed.

Description This issue is introduced in or before Commit-4. Adding 1 in line 832 in listing 2.42 cannot increase the calculation precision as `self.gas_compensation_ratio` is rather large.

```
820    #[payable]
821    pub fn liquidation(&mut self, account: AccountId) {
822        ...
823        if ratio >= INIT_NO_LIQUIDATION_FEE_RATE {
824            liquidation_fee = _allot_debt
825                            .checked_mul(self.liquidation_fee_ratio).expect(ERR_MUL)
826                            .checked_mul(STAKE_RATIO_BASE).expect(ERR_MUL)//16
827                            .checked_div(self.token_price).expect(ERR_DIV);
828        }else{
829            allot_ratio = ratio
830                .checked_sub(self.gas_compensation_ratio).expect(ERR_SUB)
831                .checked_add(1).expect(ERR_ADD);
832        }
833        ...
```

Listing 2.42: liquidation:lib.rs

Suggestion I Remove the added "1" in line 831 of listing 2.42.

2.3.12 The Risk of Centralized Design

Status Acknowledged

Description Description The project has a highly centralized design. **The contract owner has very high privilege that can add/delete the multi-signature managers and can withdraw the liquidation fee and reward, etc.** Such mechanism is absolutely centralized, which has a complete control power over all tokens. We highly suggest that the project owner should enforce security mechanisms to protect the private keys of the contract owner to manage the contracts.