BLOCKSEC

Security Audit
Report for DeltaTrade

Date: April 25, 2024 **Version:** 2.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	DeltaTrade
Target	DeltaTrade

Version History

Version	Date	Description
Version1	February 23, 2024	First Release
Version2	April 25, 2024	Second Release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2024-04-26 11:12:20+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

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

The target of this audit is the code repository of DeltaTrade[1].

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

Project	Version	Commit Hash
DeltaTrade	`Version 1`	`49b40d37822a28b866426c9376fa8cdc43106550`
	`Version 2`	`69f3c03b74a0b6daccfda1e94314e063dcd89ef6`
	`Version 3`	`38da7f16159adc6778078bfb27139dae03d050c8`

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

[1] https://github.com/DeltaBotDev/Contracts

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

We show the main concrete checkpoints in the following.

1.3.1 Software Security

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

1.3.2 DeFi Security

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

1.3.3 NFT Security

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

1.3.4 Additional Recommendation

* Gas optimization

* Code quality and style

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

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

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

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

Chapter 2 Findings

In total, we find **twenty-six** potential issues. Besides, we also have **four** recommendations and **three** notes as follows:

- High Risk: 10
- Medium Risk: 9
- Low Risk: 7
- Recommendation: 4
- Note: 3

ID	Severity	Description	Category	Status
1	Medium	Incorrect Error Message in Function create_bot()	DeFi Security	Fixed
2	High	Incorrect Target Address of Callback Function	DeFi Security	Fixed
3	Low	Lack of Storage Release	DeFi Security	Fixed
4	Medium	Lack of Attached Transfer Fee	DeFi Security	Confirmed
5	Low	Lack of Check for the Parameter valid_until_time	DeFi Security	Fixed
6	Low	Lack of Check for the Parameter slippage	DeFi Security	Confirmed
7	Medium	Unrefunded Storage Fee	DeFi Security	Fixed
8	Medium	Lack of Attached Storage Fee in Function add_refer()	DeFi Security	Fixed
9	Medium	Inappropriate Refund Mechanisms	DeFi Security	Confirmed
10	High	Incorrect refund balance in Function after_wrap_near_for_create_bot()	DeFi Security	Fixed
11	High	Lack of Check in function close_bot()	DeFi Security	Fixed
12	High	Lack of State Rollback in Callback Function	DeFi Security	Confirmed
13	Low	Redundant Refund Logic in Function internal_check_bot_amount()	DeFi Security	Fixed
14	High	Lack of Proper Handling of Token Decimals	DeFi Security	Fixed
15	Low	Gas Waste due to Redundant Checks in Function internal_create_bot()	DeFi Security	Confirmed
16	Medium	Unreasonable Logic in Function internal_check_near_amount()	DeFi Security	Confirmed
17	Low	Incorrect Revenue Token Returned in Forward Order	DeFi Security	Fixed
18	High	Function token_storage_deposit() Fails to Deposit Storage Fees	DeFi Security	Fixed

19	Low	Lack of Check on Parameter in Function token_storage_deposit()	DeFi Security	Fixed
20	High	Incorrect Storage_Key in Function internal_add_refer_recommend_user()	DeFi Security	Fixed
21	Medium	Unrefunded Near Due to WNEAR is Not in Whitelist	DeFi Security	Fixed
22	High	Incorrect Storage Fee Logic (I)	DeFi Security	Fixed
23	High	Incorrect Storage Fee Logic (II)	DeFi Security	Fixed
24	Medium	Incorrect Logic in Function internal_check_near_amount()	DeFi Security	Fixed
25	Medium	Lack of Storage Fee in Function taker_orders()	DeFi Security	Fixed
26	High	Grid_Bot Will Never Start Due to Incorrect Parameters	DeFi Security	Fixed
27	-	Redundant Code	Recommendation	Fixed
28	-	Redundant Implementation of NEAR Transfer	Recommendation	Fixed
29	-	Lack of Minimum Value Check for taker_order.amount_sell	Recommendation	Fixed
30	-	Lack of Check Parameter in Function set_refer_fee_rate()	Recommendation	Fixed
31	-	Centralization Risks	Note	
32	-	Delayed Activation of grid_bot Due to Volatile Price Fluctuations	Note	
33	-	Storage Usage for Token Never Released	Note	

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Incorrect Error Message in Function create_bot()

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Function `create_bot()` requires that the total sum of orders is less than the specified `MAX_GRID_COUNT` (line 40). Otherwise, an error will be thrown. However, the error message will be displayed as "`PAUSE_OR_SHUTDOWN`", which is incorrect. The above issue also occurs in line 50.

```
34  pub fn create_bot(&mut self, name: String, pair_id: String, slippage: u16, grid_type: GridType
        ,
35                  grid_rate: u16, grid_offset: U128, first_base_amount: U128, first_quote_amount
                        : U128,
36                  last_base_amount: U128, last_quote_amount: U128, fill_base_or_quote: bool,
                        grid_sell_count: u16, grid_buy_count: u16,
37                  trigger_price: U128, take_profit_price: U128, stop_loss_price: U128,
                        valid_until_time: U128,
38                  entry_price: U128) {
39      let grid_offset_256 = U256C::from(grid_offset.0);
40      let first_base_amount_256 = U256C::from(first_base_amount.0);
41      let first_quote_amount_256 = U256C::from(first_quote_amount.0);
42      let last_base_amount_256 = U256C::from(last_base_amount.0);
43      let last_quote_amount_256 = U256C::from(last_quote_amount.0);
44      let trigger_price_256 = U256C::from(trigger_price.0);
45      let take_profit_price_256 = U256C::from(take_profit_price.0);
46      let stop_loss_price_256 = U256C::from(stop_loss_price.0);
47      let valid_until_time_256 = U256C::from(valid_until_time.0);
48      let entry_price_256 = U256C::from(entry_price.0);
49
50      require!(valid_until_time.0 > env::block_timestamp_ms() as u128, INVALID_UNTIL_TIME);
51
52      require!(self.pair_map.contains_key(&pair_id), INVALID_PAIR_ID);
53      let pair = self.pair_map.get(&pair_id).unwrap().clone();
54      let user = env::predecessor_account_id();
55
56      // require!(self.status == GridStatus::Running, PAUSE_OR_SHUTDOWN);
57      if self.status != GridStatus::Running {
58          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
59          return;
60      }
61
62      if grid_buy_count + grid_sell_count > MAX_GRID_COUNT {
63          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                MORE_THAN_MAX_GRID_COUNT);
```

```
64          return;
65      }
66
67      // calculate all assets
68      let (base_amount_sell, quote_amount_buy) = GridBotContract::internal_calculate_bot_assets(
            first_quote_amount_256.clone(), last_base_amount_256.clone(), grid_sell_count.clone(),
             grid_buy_count.clone(),
69                                                  grid_type.clone(), grid_rate.clone(),
                                                        grid_offset_256.clone(), fill_base_or_quote.
                                                        clone());
70
71      // require!(env::attached_deposit() >= STORAGE_FEE, LESS_STORAGE_FEE);
72      if !self.internal_check_near_amount(&user, &pair, env::attached_deposit(), base_amount_sell
            , quote_amount_buy) {
73          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                INVALID_AMOUNT);
74          return;
75      }
76      // last_quote_amount / last_base_amount > first_quote_amount > first_base_amount
77      // amount must u128, u128 * u128 <= u256, so, it's ok
78      let (result, reason) = self.internal_check_bot_amount(grid_sell_count, grid_buy_count,
            first_base_amount_256, first_quote_amount_256,
79                                                            last_base_amount_256, last_quote_amount_256,
                                                                 &user, &pair, base_amount_sell,
                                                                quote_amount_buy);
80      if !result {
81          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(), &
                reason);
82          return;
83      }
84
85      // create bot
86      let mut new_grid_bot = GridBot {name, active: false, user: user.clone(), bot_id: "".
            to_string(), closed: false, pair_id, grid_type,
87          grid_sell_count: grid_sell_count.clone(), grid_buy_count: grid_buy_count.clone(),
                grid_rate, grid_offset: grid_offset_256,
88          first_base_amount: first_base_amount_256, first_quote_amount: first_quote_amount_256,
                last_base_amount: last_base_amount_256,
89          last_quote_amount: last_quote_amount_256, fill_base_or_quote, trigger_price:
                trigger_price_256, trigger_price_above_or_below: false,
90          take_profit_price: take_profit_price_256, stop_loss_price: stop_loss_price_256,
                valid_until_time: valid_until_time_256,
91          total_quote_amount: quote_amount_buy, total_base_amount: base_amount_sell, revenue:
                U256C::from(0), total_revenue: U256C::from(0)
92      };
93
94      if self.internal_need_wrap_near(&user, &pair, base_amount_sell, quote_amount_buy) {
95          // wrap near to wnear first
96          self.deposit_near_to_get_wnear_for_create_bot(&pair, &user, slippage, &entry_price_256,
                 &mut new_grid_bot, env::attached_deposit() - STORAGE_FEE);
97      } else {
98          // request token price
99          self.get_price_for_create_bot(&pair, &user, slippage, &entry_price_256, &mut
```

```
                new_grid_bot);
100         }
101     }
```

Listing 2.1: grid_bot.rs

```
 2
 3   pub const PAUSE_OR_SHUTDOWN: &str = "PAUSE_OR_SHUTDOWN";
```

Listing 2.2: errors.rs

Impact Incorrect error messages may mislead users.

Suggestion Return the correct error messages.

2.1.2 Incorrect Target Address of Callback Function

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Function `withdraw_unowned_asset()` initiates a cross-contract invocation to query the token balance and executes the refund logic accordingly. However, when invoking the callback function `after_ft_balance_of_for_withdraw_unowned_asset()`, the target contract address is set as `owner.id`, which is incorrect.

```
190     pub fn withdraw_unowned_asset(&mut self, token: AccountId, to_user: AccountId) {
191         self.assert_owner();
192         Promise::new(token.clone())
193             .function_call(
194                 "ft_balance_of".to_string(),
195                 json!({"account_id": env::current_account_id()}).to_string().into_bytes(),
196                 0,
197                 Gas(0),
198             )
199             .then(
200                 Self::ext(self.owner_id.clone())
201                     .with_static_gas(GAS_FOR_AFTER_FT_TRANSFER)
202                     .after_ft_balance_of_for_withdraw_unowned_asset(
203                         token.clone(),
204                         to_user,
205                     )
206             );
207     }
```

Listing 2.3: grid_bot.rs

Impact Assets can not be withdrawn.

Suggestion Replace `self.owner_id` with `current_account_id()`.

2.1.3 Lack of Storage Release

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Function `internal_reduce_asset()` is used to reduce and update the balance of the corresponding tokens for the user. When a user's balance of a specific token is reduced to 0, it will still be stored in the corresponding data structure (`user_balances_map`), which is a waste of storage. The above issue also occurs in the function `internal_reduce_refer_fee()`.

```
11  pub fn internal_reduce_asset(&mut self, user: &AccountId, token: &AccountId, amount: &U256C) {
12      let mut user_balances = self.user_balances_map.get(user).unwrap_or_else(|| {
13          let mut map = LookupMap::new(StorageKey::UserBalanceSubKey(user.clone()));
14          map.insert(token, &U256C::from(0));
15          map
16      });
17
18
19      let balance = user_balances.get(token).unwrap_or(U256C::from(0));
20      user_balances.insert(token, &(balance - amount));
21
22
23      self.user_balances_map.insert(user, &user_balances);
24  }
```

Listing 2.4: grid_bot_asset.rs

```
355  pub fn internal_reduce_refer_fee(&mut self, user: &AccountId, token: &AccountId, amount: &U128
         ) {
356      if amount.0 == 0 {
357          return;
358      }
359      if !self.refer_fee_map.contains_key(user) {
360          self.refer_fee_map.insert(user, &LookupMap::new(StorageKey::ReferFeeSubKey(user.clone()
                 )));
361      }
362      let mut tokens_map = self.refer_fee_map.get(user).unwrap();
363      require!(tokens_map.contains_key(token), INVALID_TOKEN);
364      tokens_map.insert(token, &U128::from(tokens_map.get(token).unwrap().0 - amount.clone().0));
365      self.refer_fee_map.insert(user, &tokens_map);
366  }
```

Listing 2.5: grid_bot_asset.rs

Impact Storage is wasted when the token balance reaches zero.

Suggestion Check if the user's token balance is zero, if so, remove the related `key-value` data.

2.1.4 Lack of Attached Transfer Fee

Severity Medium

Status Confirmed

Introduced by Version 1

Description Users can withdraw their `revenue` through the function `claim()`, which will transfer the withdrawal `NEP-141` token to the user. 1 `yocto NEAR` is attached when invoking the function `ft_transfer()` and `near_withdraw()`. However, the function `claim()` does not require the user to attach this fee, which is incorrect. The above issue also occurs in function `internal_create_bot_refund_with_near()`.

```
109   pub fn claim(&mut self, bot_id: String) {
110       require!(self.bot_map.contains_key(&bot_id), BOT_NOT_EXIST);
111       let mut bot = self.bot_map.get(&bot_id).unwrap().clone();
112       let pair = self.pair_map.get(&(bot.pair_id)).unwrap().clone();
113       // harvest revenue
114       let (revenue_token, revenue) = self.internal_harvest_revenue(&mut bot, &pair);
115       self.internal_withdraw(&(bot.user), &revenue_token, revenue);
116       self.bot_map.insert(&bot_id, &bot);
117       // event
118       emit::claim(&env::predecessor_account_id(), &(bot.user), bot_id, &revenue_token, revenue);
119   }
```

Listing 2.6: grid_bot.rs

```
216   pub fn internal_withdraw(&mut self, user: &AccountId, token: &AccountId, amount: U256C) {
217       if amount.as_u128() == 0 {
218           return;
219       }
220       // reduce user asset
221       self.internal_reduce_asset(user, token, &amount);
222       if token.clone() == self.wnear {
223           // wrap to near
224           self.withdraw_near(user, amount.as_u128());
225       } else {
226           // start transfer
227           self.internal_ft_transfer(user, token, amount.as_u128());
228       }
229       emit::withdraw_started(user, amount.as_u128(), token);
230   }
```

Listing 2.7: grid_bot_asset.rs

```
16   pub fn withdraw_near(&mut self, user: &AccountId, amount: u128) {
17       ext_wnear::ext(self.wnear.clone())
18           .with_attached_deposit(1)
19           .near_withdraw(U128::from(amount))
20           .then(
21               Self::ext(env::current_account_id())
22                   .after_withdraw_near(
23                       user,
24                       amount,
25                   )
26           );
27   }
```

Listing 2.8: wnear.rs

```
57  pub fn internal_ft_transfer(&mut self, account_id: &AccountId, token_id: &AccountId, amount:
        Balance) -> Promise {
58      ext_fungible_token::ext(token_id.clone())
59          .with_attached_deposit(ONE_YOCTO)
60          .with_static_gas(GAS_FOR_FT_TRANSFER)
61          .ft_transfer(
62              account_id.clone(),
63              amount.into(),
64              None,
65          ).then(
66          Self::ext(env::current_account_id())
67              .with_static_gas(GAS_FOR_AFTER_FT_TRANSFER)
68              .after_ft_transfer(
69                  account_id.clone(),
70                  token_id.clone(),
71                  amount.into(),
72              )
73      )
74  }
```

Listing 2.9: token.rs

```
299  pub fn internal_create_bot_refund_with_near(&mut self, user: &AccountId, pair: &Pair,
         near_amount: u128, reason: &str) {
300      self.internal_create_bot_refund(user, pair, reason);
301      self.internal_near_refund(user, near_amount);
302  }
```

Listing 2.10: grid_bot_asset.rs

```
304  pub fn internal_create_bot_refund(&mut self, user: &AccountId, pair: &Pair, reason: &str) {
305      self.internal_withdraw_all(user, &pair.base_token);
306      self.internal_withdraw_all(user, &pair.quote_token);
307      emit::create_bot_error(user, reason);
308  }
```

Listing 2.11: grid_bot_asset.rs

```
256  pub fn internal_withdraw_all(&mut self, user: &AccountId, token: &AccountId) {
257      let balance = self.internal_get_user_balance(user, token);
258      self.internal_withdraw(user, token, balance);
259  }
```

Listing 2.12: grid_bot_asset.rs

Impact The contract account can run out of storage fees, potentially leading to a DoS situation.

Suggestion Use the attribute `#[payable]` to annotate the function `claim()`, and add a check to ensure 1 `yocto NEAR` is attached.

Feedback The function `claim()` now incorporates a check to ensure that the `attached NEAR` is adequate for the transfer fee of 1 `yocto NEAR`. The refund mechanism for failed `grid_bot` creations remains unchanged. Since 1 `yocto NEAR` is negligible, and `STORAGE_FEE` of 0.01 `NEAR` is charged for each `grid_bot` creation, this amount is sufficient to cover the transfer fee incurred by multiple refunds.

2.1.5 Lack of Check for the Parameter valid_until_time

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Function `create_bot()` receives several parameters, including `valid_until_time`, which is used to specify the expiration time of the `grid_bot`. However, the function does not check whether this time is earlier than the current `block.timestamp`. The `grid_bot` may expire immediately if the `valid_until_time` is less than the `block.timestamp`.

```
13  pub fn create_bot(&mut self, name: String, pair_id: String, slippage: u16, grid_type: GridType
        ,
14                  grid_rate: u16, grid_offset: U128, first_base_amount: U128, first_quote_amount
                        : U128,
15                  last_base_amount: U128, last_quote_amount: U128, fill_base_or_quote: bool,
                        grid_sell_count: u16, grid_buy_count: u16,
16                  trigger_price: U128, take_profit_price: U128, stop_loss_price: U128,
                        valid_until_time: U128,
17                  entry_price: U128) {
18      let grid_offset_256 = U256C::from(grid_offset.0);
19      let first_base_amount_256 = U256C::from(first_base_amount.0);
20      let first_quote_amount_256 = U256C::from(first_quote_amount.0);
21      let last_base_amount_256 = U256C::from(last_base_amount.0);
22      let last_quote_amount_256 = U256C::from(last_quote_amount.0);
23      let trigger_price_256 = U256C::from(trigger_price.0);
24      let take_profit_price_256 = U256C::from(take_profit_price.0);
25      let stop_loss_price_256 = U256C::from(stop_loss_price.0);
26      let valid_until_time_256 = U256C::from(valid_until_time.0);
27      let entry_price_256 = U256C::from(entry_price.0);
28
29      require!(valid_until_time.0 > env::block_timestamp_ms() as u128, INVALID_UNTIL_TIME);
30
31      require!(self.pair_map.contains_key(&pair_id), INVALID_PAIR_ID);
32      let pair = self.pair_map.get(&pair_id).unwrap().clone();
33      let user = env::predecessor_account_id();
34
35      // require!(self.status == GridStatus::Running, PAUSE_OR_SHUTDOWN);
36      if self.status != GridStatus::Running {
37          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
38          return;
39      }
40
41      if grid_buy_count + grid_sell_count > MAX_GRID_COUNT {
```

```
42          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                MORE_THAN_MAX_GRID_COUNT);
43          return;
44      }
45
46      // calculate all assets
47      let (base_amount_sell, quote_amount_buy) = GridBotContract::internal_calculate_bot_assets(
            first_quote_amount_256.clone(), last_base_amount_256.clone(), grid_sell_count.clone(),
             grid_buy_count.clone(),
48                                                  grid_type.clone(), grid_rate.clone(),
                                                        grid_offset_256.clone(), fill_base_or_quote.
                                                        clone());
49
50      // require!(env::attached_deposit() >= STORAGE_FEE, LESS_STORAGE_FEE);
51      if !self.internal_check_near_amount(&user, &pair, env::attached_deposit(), base_amount_sell
            , quote_amount_buy) {
52          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                INVALID_AMOUNT);
53          return;
54      }
55      // last_quote_amount / last_base_amount > first_quote_amount > first_base_amount
56      // amount must u128, u128 * u128 <= u256, so, it's ok
57      let (result, reason) = self.internal_check_bot_amount(grid_sell_count, grid_buy_count,
            first_base_amount_256, first_quote_amount_256,
58                                                            last_base_amount_256, last_quote_amount_256,
                                                                 &user, &pair, base_amount_sell,
                                                                quote_amount_buy);
59      if !result {
60          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(), &
                reason);
61          return;
62      }
63
64      // create bot
65      let mut new_grid_bot = GridBot {name, active: false, user: user.clone(), bot_id: "".
            to_string(), closed: false, pair_id, grid_type,
66          grid_sell_count: grid_sell_count.clone(), grid_buy_count: grid_buy_count.clone(),
                grid_rate, grid_offset: grid_offset_256,
67          first_base_amount: first_base_amount_256, first_quote_amount: first_quote_amount_256,
                last_base_amount: last_base_amount_256,
68          last_quote_amount: last_quote_amount_256, fill_base_or_quote, trigger_price:
                trigger_price_256, trigger_price_above_or_below: false,
69          take_profit_price: take_profit_price_256, stop_loss_price: stop_loss_price_256,
                valid_until_time: valid_until_time_256,
70          total_quote_amount: quote_amount_buy, total_base_amount: base_amount_sell, revenue:
                U256C::from(0), total_revenue: U256C::from(0)
71      };
72
73      if self.internal_need_wrap_near(&user, &pair, base_amount_sell, quote_amount_buy) {
74          // wrap near to wnear first
75          self.deposit_near_to_get_wnear_for_create_bot(&pair, &user, slippage, &entry_price_256,
                &mut new_grid_bot, env::attached_deposit() - STORAGE_FEE);
76      } else {
```

```
77          // request token price
78          self.get_price_for_create_bot(&pair, &user, slippage, &entry_price_256, &mut
                new_grid_bot);
79      }
80  }
```

Listing 2.13: grid_bot.rs

```
 8  pub fn query_order(&self, bot_id: String, forward_or_reverse: bool, level: usize) -> (Order,
        bool) {
 9      require!(self.order_map.contains_key(&bot_id), INVALID_BOT_ID);
10      require!(self.bot_map.contains_key(&bot_id), INVALID_BOT_ID);
11      let bot = self.bot_map.get(&bot_id).unwrap();
12      require!(!(bot.closed.clone()), bot.bot_id.clone() + BOT_CLOSED);
13      require!(bot.active.clone(), bot.bot_id.clone() + BOT_DISABLE);
14      require!(self.pair_map.contains_key(&(bot.pair_id.clone())), INVALID_PAIR_ID);
15      // check timestamp
16      require!(bot.valid_until_time >= U256C::from(env::block_timestamp_ms()), BOT_EXPIRED);
17      let bot_orders = self.order_map.get(&bot_id).unwrap();
18      let orders = if forward_or_reverse {
19          bot_orders.forward_orders
20      } else {
21          bot_orders.reverse_orders
22      };
23      // check order
24      require!(orders.get(level.clone() as u64).is_some(), INVALID_PARAM);
25      let order = &orders.get(level as u64).unwrap();
26      if GridBotContract::internal_order_is_empty(order) {
27          require!(forward_or_reverse, INVALID_FORWARD_OR_REVERSE);
28          // The current grid order has not been placed yet
29          let pair = self.pair_map.get(&(bot.pair_id.clone())).unwrap();
30          return ((GridBotContract::internal_get_first_forward_order(bot.clone(), pair.clone(),
                level.clone())), false);
31      }
32      return (order.clone(), true);
33  }
```

Listing 2.14: orderbook_view.rs

Impact The created `grid_bot` may expire immediately.

Suggestion Add a check to ensure that the `valid_until_time` is greater than the current `block.timestamp`.

2.1.6 Lack of Check for the Parameter slippage

Severity Low

Status Confirmed

Introduced by `Version 1`

Description During the creation of a `grid_bot`, users provide the parameters `entry_price` and `slippage`, which are used to control the `slippage` based on the valid price obtained from the

oracle. However, the slippage is allowed to be set to 0, which can result in a high probability of creation failures.

```
13  pub fn create_bot(&mut self, name: String, pair_id: String, slippage: u16, grid_type: GridType
        ,
14                    grid_rate: u16, grid_offset: U128, first_base_amount: U128, first_quote_amount
                          : U128,
15                    last_base_amount: U128, last_quote_amount: U128, fill_base_or_quote: bool,
                          grid_sell_count: u16, grid_buy_count: u16,
16                    trigger_price: U128, take_profit_price: U128, stop_loss_price: U128,
                          valid_until_time: U128,
17                    entry_price: U128) {
18      let grid_offset_256 = U256C::from(grid_offset.0);
19      let first_base_amount_256 = U256C::from(first_base_amount.0);
20      let first_quote_amount_256 = U256C::from(first_quote_amount.0);
21      let last_base_amount_256 = U256C::from(last_base_amount.0);
22      let last_quote_amount_256 = U256C::from(last_quote_amount.0);
23      let trigger_price_256 = U256C::from(trigger_price.0);
24      let take_profit_price_256 = U256C::from(take_profit_price.0);
25      let stop_loss_price_256 = U256C::from(stop_loss_price.0);
26      let valid_until_time_256 = U256C::from(valid_until_time.0);
27      let entry_price_256 = U256C::from(entry_price.0);
28
29      require!(valid_until_time.0 > env::block_timestamp_ms() as u128, INVALID_UNTIL_TIME);
30
31      require!(self.pair_map.contains_key(&pair_id), INVALID_PAIR_ID);
32      let pair = self.pair_map.get(&pair_id).unwrap().clone();
33      let user = env::predecessor_account_id();
34
35      // require!(self.status == GridStatus::Running, PAUSE_OR_SHUTDOWN);
36      if self.status != GridStatus::Running {
37          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
38          return;
39      }
40
41      if grid_buy_count + grid_sell_count > MAX_GRID_COUNT {
42          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                MORE_THAN_MAX_GRID_COUNT);
43          return;
44      }
45
46      // calculate all assets
47      let (base_amount_sell, quote_amount_buy) = GridBotContract::internal_calculate_bot_assets(
            first_quote_amount_256.clone(), last_base_amount_256.clone(), grid_sell_count.clone(),
             grid_buy_count.clone(),
48                                                  grid_type.clone(), grid_rate.clone(),
                                                        grid_offset_256.clone(), fill_base_or_quote.
                                                        clone());
49
50      // require!(env::attached_deposit() >= STORAGE_FEE, LESS_STORAGE_FEE);
51      if !self.internal_check_near_amount(&user, &pair, env::attached_deposit(), base_amount_sell
            , quote_amount_buy) {
```

```
52          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                INVALID_AMOUNT);
53          return;
54      }
55      // last_quote_amount / last_base_amount > first_quote_amount > first_base_amount
56      // amount must u128, u128 * u128 <= u256, so, it's ok
57      let (result, reason) = self.internal_check_bot_amount(grid_sell_count, grid_buy_count,
            first_base_amount_256, first_quote_amount_256,
58                                                  last_base_amount_256, last_quote_amount_256,
                                                        &user, &pair, base_amount_sell,
                                                    quote_amount_buy);
59      if !result {
60          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(), &
                reason);
61          return;
62      }
63
64      // create bot
65      let mut new_grid_bot = GridBot {name, active: false, user: user.clone(), bot_id: "".
            to_string(), closed: false, pair_id, grid_type,
66          grid_sell_count: grid_sell_count.clone(), grid_buy_count: grid_buy_count.clone(),
                grid_rate, grid_offset: grid_offset_256,
67          first_base_amount: first_base_amount_256, first_quote_amount: first_quote_amount_256,
                last_base_amount: last_base_amount_256,
68          last_quote_amount: last_quote_amount_256, fill_base_or_quote, trigger_price:
                trigger_price_256, trigger_price_above_or_below: false,
69          take_profit_price: take_profit_price_256, stop_loss_price: stop_loss_price_256,
                valid_until_time: valid_until_time_256,
70          total_quote_amount: quote_amount_buy, total_base_amount: base_amount_sell, revenue:
                U256C::from(0), total_revenue: U256C::from(0)
71      };
72
73      if self.internal_need_wrap_near(&user, &pair, base_amount_sell, quote_amount_buy) {
74          // wrap near to wnear first
75          self.deposit_near_to_get_wnear_for_create_bot(&pair, &user, slippage, &entry_price_256,
                &mut new_grid_bot, env::attached_deposit() - STORAGE_FEE);
76      } else {
77          // request token price
78          self.get_price_for_create_bot(&pair, &user, slippage, &entry_price_256, &mut
                new_grid_bot);
79      }
80  }
```

Listing 2.15: grid_bot.rs

```
15      pub fn internal_create_bot(&mut self,
16                             base_price: Price,
17                             quote_price: Price,
18                             user: &AccountId,
19                             slippage: u16,
20                             entry_price: &U256C,
21                             pair: &Pair,
22                             grid_bot: &mut GridBot) -> bool {
```

```
23      if self.status != GridStatus::Running {
24          self.internal_create_bot_refund(&user, &pair, PAUSE_OR_SHUTDOWN);
25          return false;
26      }
27      // require!(self.internal_check_oracle_price(*entry_price, base_price.clone(), quote_price.
            clone(), slippage), INVALID_PRICE);
28      if !self.internal_check_oracle_price(*entry_price, base_price.clone(), quote_price.clone(),
             slippage) {
29          self.internal_create_bot_refund(user, pair, INVALID_PRICE);
30          return false;
31      }
32      // check balance
33      // require!(self.internal_get_user_balance(user, &(pair.base_token)) >= base_amount_sell,
            LESS_BASE_TOKEN);
34      if self.internal_get_user_balance(user, &(pair.base_token)) < grid_bot.total_base_amount {
35          self.internal_create_bot_refund(user, pair, LESS_BASE_TOKEN);
36          return false;
37      }
38      // require!(self.internal_get_user_balance(user, &(pair.quote_token)) >= quote_amount_buy,
            LESS_QUOTE_TOKEN);
39      if self.internal_get_user_balance(user, &(pair.quote_token)) < grid_bot.total_quote_amount
            {
40          self.internal_create_bot_refund(user, pair, LESS_QUOTE_TOKEN);
41          return false;
42      }
43
44
45      // create bot id
46      let next_bot_id = format!("GRID:{}", self.internal_get_and_use_next_bot_id().to_string());
47      grid_bot.bot_id = next_bot_id;
48
49
50      // initial orders space, create empty orders
51      let grid_count = grid_bot.grid_sell_count.clone() + grid_bot.grid_buy_count.clone();
52      self.create_default_orders(grid_bot.bot_id.clone(), grid_count);
53
54
55      // transfer assets
56      self.internal_transfer_assets_to_lock(&user, &pair.base_token, grid_bot.total_base_amount);
57      self.internal_transfer_assets_to_lock(&user, &pair.quote_token, grid_bot.total_quote_amount
            );
58
59
60      // init active status of bot
61      self.internal_init_bot_status(grid_bot, entry_price);
62
63
64      // insert bot
65      self.bot_map.insert(&(grid_bot.bot_id), &grid_bot);
66
67
68      emit::create_bot(&grid_bot.user, grid_bot.bot_id.clone(), base_price.price.0.to_string(),
            quote_price.price.0.to_string(), base_price.expo.to_string(), quote_price.expo.
```

```
            to_string());
69      return true;
70  }
```

Listing 2.16: grid_bot_internal.rs

```
 9  pub fn internal_check_oracle_price(&self, entry_price: U256C, base_price: Price, quote_price:
        Price, slippage: u16) -> bool {
10      if base_price.publish_time as u64 * 1000 + self.oracle_valid_time.clone() < env::
            block_timestamp_ms() {
11          return false;
12      }
13      if quote_price.publish_time as u64 * 1000 + self.oracle_valid_time.clone() < env::
            block_timestamp_ms() {
14          return false;
15      }
16      let oracle_pair_price = (BigDecimal::from(base_price.price.0 as u64) / BigDecimal::from(
            quote_price.price.0 as u64) * BigDecimal::from(PRICE_DENOMINATOR)).round_down_u128();
17
18
19      if entry_price.as_u128() >= oracle_pair_price {
20          return (entry_price.as_u128() - oracle_pair_price) * SLIPPAGE_DENOMINATOR as u128 /
                entry_price.as_u128() <= slippage as u128;
21      } else {
22          return (oracle_pair_price - entry_price.as_u128()) * SLIPPAGE_DENOMINATOR as u128 /
                entry_price.as_u128() <= slippage as u128;
23      }
24  }
```

Listing 2.17: grid_bot_check.rs

Impact Allowing a `slippage` value of 0 can increase the probability of creation failures, resulting in wasted gas.

Suggestion Add a check to ensure the `slippage` is not zero when creating the `grid_bot`.

Feedback The protocol allows users to set the `slippage` to zero.

2.1.7 Unrefunded Storage Fee

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Extra storage fee is required when creating a `grid_bot`. However, when refunding the funds to users if the bot creation fails, the storage fee is not refunded. Specifically, the function will invoke the function `get_price_for_create_bot()` to fetch the prices of the tokens from the oracle, and check in the callback function `get_price_for_create_bot_callback()` whether both token prices have been retrieved. If not, the contract will invoke the function `internal_create_bot_refund()` to refund the user's funds. The storage fee is not included. The above issue also occurs in the functions `internal_create_bot()` and `after_wrap_near_for_create_bot()`.

```
13  pub fn create_bot(&mut self, name: String, pair_id: String, slippage: u16, grid_type: GridType
        ,
14                  grid_rate: u16, grid_offset: U128, first_base_amount: U128, first_quote_amount:
                        U128,
15                  last_base_amount: U128, last_quote_amount: U128, fill_base_or_quote: bool,
                        grid_sell_count: u16, grid_buy_count: u16,
16                  trigger_price: U128, take_profit_price: U128, stop_loss_price: U128,
                        valid_until_time: U128,
17                  entry_price: U128) {
18      let grid_offset_256 = U256C::from(grid_offset.0);
19      let first_base_amount_256 = U256C::from(first_base_amount.0);
20      let first_quote_amount_256 = U256C::from(first_quote_amount.0);
21      let last_base_amount_256 = U256C::from(last_base_amount.0);
22      let last_quote_amount_256 = U256C::from(last_quote_amount.0);
23      let trigger_price_256 = U256C::from(trigger_price.0);
24      let take_profit_price_256 = U256C::from(take_profit_price.0);
25      let stop_loss_price_256 = U256C::from(stop_loss_price.0);
26      let valid_until_time_256 = U256C::from(valid_until_time.0);
27      let entry_price_256 = U256C::from(entry_price.0);
28
29
30      require!(self.pair_map.contains_key(&pair_id), INVALID_PAIR_ID);
31      let pair = self.pair_map.get(&pair_id).unwrap().clone();
32      let user = env::predecessor_account_id();
33
34
35      // require!(self.status == GridStatus::Running, PAUSE_OR_SHUTDOWN);
36      if self.status != GridStatus::Running {
37          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
38          return;
39      }
40
41
42      if grid_buy_count + grid_sell_count > MAX_GRID_COUNT {
43          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
44          return;
45      }
46
47
48      // calculate all assets
49      let (base_amount_sell, quote_amount_buy) = GridBotContract::internal_calculate_bot_assets(
            first_quote_amount_256.clone(), last_base_amount_256.clone(), grid_sell_count.clone(),
             grid_buy_count.clone(),
50                                                  grid_type.clone(), grid_rate.clone(),
                                                        grid_offset_256.clone(), fill_base_or_quote.
                                                        clone());
51
52
53      // require!(env::attached_deposit() >= STORAGE_FEE, LESS_STORAGE_FEE);
54      if !self.internal_check_near_amount(&user, &pair, env::attached_deposit(), base_amount_sell
```

```
            , quote_amount_buy) {
55          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
56          return;
57      }
58      // last_quote_amount / last_base_amount > first_quote_amount > first_base_amount
59      // amount must u128, u128 * u128 <= u256, so, it's ok
60      let (result, reason) = self.internal_check_bot_amount(grid_sell_count, grid_buy_count,
            first_base_amount_256, first_quote_amount_256,
61                                                      last_base_amount_256, last_quote_amount_256,
                                                             &user, &pair, base_amount_sell,
                                                            quote_amount_buy);
62      if !result {
63          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(), &
                reason);
64          return;
65      }
66
67
68      // create bot
69      let mut new_grid_bot = GridBot {name, active: false, user: user.clone(), bot_id: "".
            to_string(), closed: false, pair_id, grid_type,
70          grid_sell_count: grid_sell_count.clone(), grid_buy_count: grid_buy_count.clone(),
                grid_rate, grid_offset: grid_offset_256,
71          first_base_amount: first_base_amount_256, first_quote_amount: first_quote_amount_256,
                last_base_amount: last_base_amount_256,
72          last_quote_amount: last_quote_amount_256, fill_base_or_quote, trigger_price:
                trigger_price_256, trigger_price_above_or_below: false,
73          take_profit_price: take_profit_price_256, stop_loss_price: stop_loss_price_256,
                valid_until_time: valid_until_time_256,
74          total_quote_amount: quote_amount_buy, total_base_amount: base_amount_sell, revenue:
                U256C::from(0), total_revenue: U256C::from(0)
75      };
76
77
78      if self.internal_need_wrap_near(&user, &pair, base_amount_sell, quote_amount_buy) {
79          // wrap near to wnear first
80          self.deposit_near_to_get_wnear_for_create_bot(&pair, &user, slippage, &entry_price_256,
                &mut new_grid_bot, env::attached_deposit() - STORAGE_FEE);
81      } else {
82          // request token price
83          self.get_price_for_create_bot(&pair, &user, slippage, &entry_price_256, &mut
                new_grid_bot);
84      }
85  }
```

Listing 2.18: grid_bot.rs

```
143 pub fn get_price_for_create_bot(
144     &mut self,
145     pair: &Pair,
146     user: &AccountId,
147     slippage: u16,
```

```
148         entry_price: &U256C,
149         grid_bot: &mut GridBot,
150     ) {
151         let (promise, tokens) = self.private_create_pair_price_request(pair);
152         promise.then(
153             Self::ext(env::current_account_id())
154                 .with_static_gas(GAS_FOR_CREATE_BOT_AFTER_ORACLE)
155                 .get_price_for_create_bot_callback(tokens.len(), tokens, user, slippage,
                        entry_price, pair, grid_bot),
156         );
157     }
```

Listing 2.19: oracle.rs

```
198     fn get_price_for_create_bot_callback(&mut self,
199                                          promise_num: usize, tokens: Vec<AccountId>, user: &AccountId,
200                                          slippage: u16, entry_price: &U256C, pair: &Pair, grid_bot: &
                                                 mut GridBot,
201     ) -> bool {
202         let price_list = self.private_get_price_list(promise_num, tokens);
203         // require!(price_list.len() == PAIR_TOKEN_LENGTH, INVALID_PAIR_PRICE_LENGTH);
204         if price_list.len() != PAIR_TOKEN_LENGTH {
205             self.internal_create_bot_refund(user, pair, INVALID_PAIR_PRICE_LENGTH);
206             return false;
207         }
208         return self.internal_create_bot(price_list[0].clone(), price_list[1].clone(), user,
                slippage, entry_price, pair, grid_bot);
209     }
```

Listing 2.20: oracle.rs

```
15      pub fn internal_create_bot(&mut self,
16                                 base_price: Price,
17                                 quote_price: Price,
18                                 user: &AccountId,
19                                 slippage: u16,
20                                 entry_price: &U256C,
21                                 pair: &Pair,
22                                 grid_bot: &mut GridBot) -> bool {
23          if self.status != GridStatus::Running {
24              self.internal_create_bot_refund(&user, &pair, PAUSE_OR_SHUTDOWN);
25              return false;
26          }
27          // require!(self.internal_check_oracle_price(*entry_price, base_price.clone(), quote_price.
                clone(), slippage), INVALID_PRICE);
28          if !self.internal_check_oracle_price(*entry_price, base_price.clone(), quote_price.clone(),
                slippage) {
29              self.internal_create_bot_refund(user, pair, INVALID_PRICE);
30              return false;
31          }
32          // check balance
33          // require!(self.internal_get_user_balance(user, &(pair.base_token)) >= base_amount_sell,
                LESS_BASE_TOKEN);
```

```
34      if self.internal_get_user_balance(user, &(pair.base_token)) < grid_bot.total_base_amount {
35          self.internal_create_bot_refund(user, pair, LESS_BASE_TOKEN);
36          return false;
37      }
38      // require!(self.internal_get_user_balance(user, &(pair.quote_token)) >= quote_amount_buy,
            LESS_QUOTE_TOKEN);
39      if self.internal_get_user_balance(user, &(pair.quote_token)) < grid_bot.total_quote_amount
            {
40          self.internal_create_bot_refund(user, pair, LESS_QUOTE_TOKEN);
41          return false;
42      }
43
44
45      // create bot id
46      let next_bot_id = format!("GRID:{}", self.internal_get_and_use_next_bot_id().to_string());
47      grid_bot.bot_id = next_bot_id;
48
49
50      // initial orders space, create empty orders
51      let grid_count = grid_bot.grid_sell_count.clone() + grid_bot.grid_buy_count.clone();
52      self.create_default_orders(grid_bot.bot_id.clone(), grid_count);
53
54
55      // transfer assets
56      self.internal_transfer_assets_to_lock(&user, &pair.base_token, grid_bot.total_base_amount);
57      self.internal_transfer_assets_to_lock(&user, &pair.quote_token, grid_bot.total_quote_amount
            );
58
59
60      // init active status of bot
61      self.internal_init_bot_status(grid_bot, entry_price);
62
63
64      // insert bot
65      self.bot_map.insert(&(grid_bot.bot_id), &grid_bot);
66
67
68      emit::create_bot(&grid_bot.user, grid_bot.bot_id.clone(), base_price.price.0.to_string(),
            quote_price.price.0.to_string(), base_price.expo.to_string(), quote_price.expo.
            to_string());
69      return true;
70  }
```

Listing 2.21: grid_bot_internal.rs

```
62  fn after_wrap_near_for_create_bot(&mut self, pair: &Pair, user: &AccountId, slippage: u16,
        entry_price: &U256C, grid_bot: &mut GridBot, amount: u128) -> bool {
63      let promise_success = is_promise_success();
64      if !promise_success.clone() {
65          // refund token and near
66          self.internal_create_bot_refund_with_near(user, pair, amount + STORAGE_FEE,
                WRAP_TO_WNEAR_ERROR);
67          emit::wrap_near_error(user, 0, amount, true);
```

```
68      } else {
69          // deposit
70          if !self.internal_deposit(&user.clone(), &self.wnear.clone(), U128::from(amount)) {
71              // maybe just need hande one token, but it's ok, no problem
72              self.internal_create_bot_refund(user, pair, WRAP_TO_WNEAR_ERROR);
73              emit::wrap_near_error(user, 0, amount, true);
74          } else {
75              // request price
76              self.get_price_for_create_bot(pair, user, slippage, entry_price, grid_bot);
77          }
78      }
79      promise_success
80  }
```

Listing 2.22: wnear.rs

Impact Users will lose the `STORAGE_FEE` when the bot creation fails.

Suggestion Add the logic to refund the storage fee in the aforementioned function.

2.1.8 Lack of Attached Storage Fee in Function add_refer()

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `DeltaBot`, the function `add_refer()` is used to add a referral relationship between a `user` and a `recommender`, which will increase the storage usage of the contract account. However, the storage fee is not charged.

```
144   pub fn add_refer(&mut self, user: AccountId, recommender: AccountId) {
145       require!(env::predecessor_account_id() == self.operator_id || env::predecessor_account_id()
              == self.owner_id, ERR_NOT_ALLOWED);
146       require!(!self.refer_user_recommender_map.contains_key(&user), ADDED_RECOMMEND);
147       require!(user != recommender, INVALID_USER);
148       self.internal_add_refer(&user, &recommender);
149   }
```

Listing 2.23: grid_bot.rs

```
144   pub fn internal_add_refer(&mut self, user: &AccountId, recommender: &AccountId) {
145       self.internal_add_refer_user_recommend(user, recommender);
146       self.internal_add_refer_recommend_user(user, recommender);
147   }
```

Listing 2.24: grid_bot_asset.rs

```
319   pub fn internal_add_refer_user_recommend(&mut self, user: &AccountId, recommender: &AccountId)
          {
320       self.refer_user_recommender_map.insert(user, recommender);
321   }
322
323
```

```
324   pub fn internal_add_refer_recommend_user(&mut self, user: &AccountId, recommender: &AccountId)
          {
325       if !self.refer_recommender_user_map.contains_key(recommender) {
326           let key = user.to_string() + ":ref_users";
327           self.refer_recommender_user_map.insert(recommender, &Vector::new(key.as_bytes().to_vec
                  ()));
328       }
329       let mut ref_users = self.refer_recommender_user_map.get(recommender).unwrap();
330       ref_users.push(user);
331
332
333       self.refer_recommender_user_map.insert(recommender, &ref_users);
334   }
```

Listing 2.25: grid_bot_asset.rs

Impact The function `add_refer()` requires storage fee, which could pose a potential DoS.

Suggestion Use the attribute `#[payable]` to annotate the function `add_refer()`, and add a check to ensure the storage fee is attached.

2.1.9 Inappropriate Refund Mechanisms

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description The current refund mechanism in the process of creating a `grid_bot` is inappropriate. Specifically, when a check fails, the contract currently refunds all previously deposited tokens instead of refunding only the tokens deposited during the current transaction, which is unreasonable.

```
13       pub fn create_bot(&mut self, name: String, pair_id: String, slippage: u16, grid_type:
             GridType,
14                 grid_rate: u16, grid_offset: U128, first_base_amount: U128, first_quote_amount
                       : U128,
15                 last_base_amount: U128, last_quote_amount: U128, fill_base_or_quote: bool,
                       grid_sell_count: u16, grid_buy_count: u16,
16                 trigger_price: U128, take_profit_price: U128, stop_loss_price: U128,
                       valid_until_time: U128,
17                 entry_price: U128) {
18       let grid_offset_256 = U256C::from(grid_offset.0);
19       let first_base_amount_256 = U256C::from(first_base_amount.0);
20       let first_quote_amount_256 = U256C::from(first_quote_amount.0);
21       let last_base_amount_256 = U256C::from(last_base_amount.0);
22       let last_quote_amount_256 = U256C::from(last_quote_amount.0);
23       let trigger_price_256 = U256C::from(trigger_price.0);
24       let take_profit_price_256 = U256C::from(take_profit_price.0);
25       let stop_loss_price_256 = U256C::from(stop_loss_price.0);
26       let valid_until_time_256 = U256C::from(valid_until_time.0);
27       let entry_price_256 = U256C::from(entry_price.0);
28
```

```
29
30      require!(self.pair_map.contains_key(&pair_id), INVALID_PAIR_ID);
31      let pair = self.pair_map.get(&pair_id).unwrap().clone();
32      let user = env::predecessor_account_id();
33
34
35      // require!(self.status == GridStatus::Running, PAUSE_OR_SHUTDOWN);
36      if self.status != GridStatus::Running {
37          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
38          return;
39      }
40
41
42      if grid_buy_count + grid_sell_count > MAX_GRID_COUNT {
43          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
44          return;
45      }
46
47
48      // calculate all assets
49      let (base_amount_sell, quote_amount_buy) = GridBotContract::internal_calculate_bot_assets(
            first_quote_amount_256.clone(), last_base_amount_256.clone(), grid_sell_count.clone(),
             grid_buy_count.clone(),
50                                                  grid_type.clone(), grid_rate.clone(),
                                                        grid_offset_256.clone(), fill_base_or_quote.
                                                        clone());
51
52
53      // require!(env::attached_deposit() >= STORAGE_FEE, LESS_STORAGE_FEE);
54      if !self.internal_check_near_amount(&user, &pair, env::attached_deposit(), base_amount_sell
            , quote_amount_buy) {
55          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
56          return;
57      }
58      // last_quote_amount / last_base_amount > first_quote_amount > first_base_amount
59      // amount must u128, u128 * u128 <= u256, so, it's ok
60      let (result, reason) = self.internal_check_bot_amount(grid_sell_count, grid_buy_count,
            first_base_amount_256, first_quote_amount_256,
61                                                            last_base_amount_256, last_quote_amount_256,
                                                                 &user, &pair, base_amount_sell,
                                                                quote_amount_buy);
62      if !result {
63          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(), &
                reason);
64          return;
65      }
66
67
68      // create bot
69      let mut new_grid_bot = GridBot {name, active: false, user: user.clone(), bot_id: "".
```

```
            to_string(), closed: false, pair_id, grid_type,
70          grid_sell_count: grid_sell_count.clone(), grid_buy_count: grid_buy_count.clone(),
                grid_rate, grid_offset: grid_offset_256,
71          first_base_amount: first_base_amount_256, first_quote_amount: first_quote_amount_256,
                last_base_amount: last_base_amount_256,
72          last_quote_amount: last_quote_amount_256, fill_base_or_quote, trigger_price:
                trigger_price_256, trigger_price_above_or_below: false,
73          take_profit_price: take_profit_price_256, stop_loss_price: stop_loss_price_256,
                valid_until_time: valid_until_time_256,
74          total_quote_amount: quote_amount_buy, total_base_amount: base_amount_sell, revenue:
                U256C::from(0), total_revenue: U256C::from(0)
75      };
76
77
78      if self.internal_need_wrap_near(&user, &pair, base_amount_sell, quote_amount_buy) {
79          // wrap near to wnear first
80          self.deposit_near_to_get_wnear_for_create_bot(&pair, &user, slippage, &entry_price_256,
                &mut new_grid_bot, env::attached_deposit() - STORAGE_FEE);
81      } else {
82          // request token price
83          self.get_price_for_create_bot(&pair, &user, slippage, &entry_price_256, &mut
                new_grid_bot);
84      }
85  }
```

Listing 2.26: grid_bot.rs

```
15  pub fn internal_create_bot(&mut self,
16                             base_price: Price,
17                             quote_price: Price,
18                             user: &AccountId,
19                             slippage: u16,
20                             entry_price: &U256C,
21                             pair: &Pair,
22                             grid_bot: &mut GridBot) -> bool {
23      if self.status != GridStatus::Running {
24          self.internal_create_bot_refund(&user, &pair, PAUSE_OR_SHUTDOWN);
25          return false;
26      }
27      // require!(self.internal_check_oracle_price(*entry_price, base_price.clone(), quote_price.
            clone(), slippage), INVALID_PRICE);
28      if !self.internal_check_oracle_price(*entry_price, base_price.clone(), quote_price.clone(),
            slippage) {
29          self.internal_create_bot_refund(user, pair, INVALID_PRICE);
30          return false;
31      }
32      // check balance
33      // require!(self.internal_get_user_balance(user, &(pair.base_token)) >= base_amount_sell,
            LESS_BASE_TOKEN);
34      if self.internal_get_user_balance(user, &(pair.base_token)) < grid_bot.total_base_amount {
35          self.internal_create_bot_refund(user, pair, LESS_BASE_TOKEN);
36          return false;
37      }
```

```
38      // require!(self.internal_get_user_balance(user, &(pair.quote_token)) >= quote_amount_buy,
            LESS_QUOTE_TOKEN);
39      if self.internal_get_user_balance(user, &(pair.quote_token)) < grid_bot.total_quote_amount {
40          self.internal_create_bot_refund(user, pair, LESS_QUOTE_TOKEN);
41          return false;
42      }
43
44
45      // create bot id
46      let next_bot_id = format!("GRID:{}", self.internal_get_and_use_next_bot_id().to_string());
47      grid_bot.bot_id = next_bot_id;
48
49
50      // initial orders space, create empty orders
51      let grid_count = grid_bot.grid_sell_count.clone() + grid_bot.grid_buy_count.clone();
52      self.create_default_orders(grid_bot.bot_id.clone(), grid_count);
53
54
55      // transfer assets
56      self.internal_transfer_assets_to_lock(&user, &pair.base_token, grid_bot.total_base_amount);
57      self.internal_transfer_assets_to_lock(&user, &pair.quote_token, grid_bot.total_quote_amount)
            ;
58
59
60      // init active status of bot
61      self.internal_init_bot_status(grid_bot, entry_price);
62
63
64      // insert bot
65      self.bot_map.insert(&(grid_bot.bot_id), &grid_bot);
66
67
68      emit::create_bot(&grid_bot.user, grid_bot.bot_id.clone(), base_price.price.0.to_string(),
            quote_price.price.0.to_string(), base_price.expo.to_string(), quote_price.expo.
            to_string());
69      return true;
70  }
```

Listing 2.27: grid_bot_internal.rs

Impact When the creation of a `grid_bot` fails, users have to deposit all the tokens again before attempting to create it once more, which is a waste of gas.

Suggestion Only refund the tokens deposited during the current transaction.

Feedback The protocol strives to minimize the retention of user assets.

2.1.10 Incorrect refund balance in Function after_wrap_near_for_create_bot()

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Function `deposit_near_to_get_wnear_for_create_bot()` deposits `NEAR` to get the `WNEAR`. In the callback function `after_wrap_near_for_create_bot()`, function `internal_deposit()` checks the `WNEAR` amount against the specified minimum amount. If the check fails, the wrapped `WNEAR` should be refunded. However, this part of `WNEAR` is not counted in the contract, resulting in an incorrect refund balance.

```
13  pub fn create_bot(&mut self, name: String, pair_id: String, slippage: u16, grid_type: GridType
        ,
14                  grid_rate: u16, grid_offset: U128, first_base_amount: U128, first_quote_amount
                        : U128,
15                  last_base_amount: U128, last_quote_amount: U128, fill_base_or_quote: bool,
                        grid_sell_count: u16, grid_buy_count: u16,
16                  trigger_price: U128, take_profit_price: U128, stop_loss_price: U128,
                        valid_until_time: U128,
17                  entry_price: U128) {
18      let grid_offset_256 = U256C::from(grid_offset.0);
19      let first_base_amount_256 = U256C::from(first_base_amount.0);
20      let first_quote_amount_256 = U256C::from(first_quote_amount.0);
21      let last_base_amount_256 = U256C::from(last_base_amount.0);
22      let last_quote_amount_256 = U256C::from(last_quote_amount.0);
23      let trigger_price_256 = U256C::from(trigger_price.0);
24      let take_profit_price_256 = U256C::from(take_profit_price.0);
25      let stop_loss_price_256 = U256C::from(stop_loss_price.0);
26      let valid_until_time_256 = U256C::from(valid_until_time.0);
27      let entry_price_256 = U256C::from(entry_price.0);
28
29
30      require!(self.pair_map.contains_key(&pair_id), INVALID_PAIR_ID);
31      let pair = self.pair_map.get(&pair_id).unwrap().clone();
32      let user = env::predecessor_account_id();
33
34
35      // require!(self.status == GridStatus::Running, PAUSE_OR_SHUTDOWN);
36      if self.status != GridStatus::Running {
37          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
38          return;
39      }
40
41
42      if grid_buy_count + grid_sell_count > MAX_GRID_COUNT {
43          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
44          return;
45      }
46
47
48      // calculate all assets
49      let (base_amount_sell, quote_amount_buy) = GridBotContract::internal_calculate_bot_assets(
            first_quote_amount_256.clone(), last_base_amount_256.clone(), grid_sell_count.clone(),
             grid_buy_count.clone(),
50                                                  grid_type.clone(), grid_rate.clone(),
```

```
                                                    grid_offset_256.clone(), fill_base_or_quote.
                                                    clone());
51
52
53      // require!(env::attached_deposit() >= STORAGE_FEE, LESS_STORAGE_FEE);
54      if !self.internal_check_near_amount(&user, &pair, env::attached_deposit(), base_amount_sell
            , quote_amount_buy) {
55          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
56          return;
57      }
58      // last_quote_amount / last_base_amount > first_quote_amount > first_base_amount
59      // amount must u128, u128 * u128 <= u256, so, it's ok
60      let (result, reason) = self.internal_check_bot_amount(grid_sell_count, grid_buy_count,
            first_base_amount_256, first_quote_amount_256,
61                                                    last_base_amount_256, last_quote_amount_256,
                                                          &user, &pair, base_amount_sell,
                                                      quote_amount_buy);
62      if !result {
63          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(), &
                reason);
64          return;
65      }
66
67
68      // create bot
69      let mut new_grid_bot = GridBot {name, active: false, user: user.clone(), bot_id: "".
            to_string(), closed: false, pair_id, grid_type,
70          grid_sell_count: grid_sell_count.clone(), grid_buy_count: grid_buy_count.clone(),
                grid_rate, grid_offset: grid_offset_256,
71          first_base_amount: first_base_amount_256, first_quote_amount: first_quote_amount_256,
                last_base_amount: last_base_amount_256,
72          last_quote_amount: last_quote_amount_256, fill_base_or_quote, trigger_price:
                trigger_price_256, trigger_price_above_or_below: false,
73          take_profit_price: take_profit_price_256, stop_loss_price: stop_loss_price_256,
                valid_until_time: valid_until_time_256,
74          total_quote_amount: quote_amount_buy, total_base_amount: base_amount_sell, revenue:
                U256C::from(0), total_revenue: U256C::from(0)
75      };
76
77
78      if self.internal_need_wrap_near(&user, &pair, base_amount_sell, quote_amount_buy) {
79          // wrap near to wnear first
80          self.deposit_near_to_get_wnear_for_create_bot(&pair, &user, slippage, &entry_price_256,
                &mut new_grid_bot, env::attached_deposit() - STORAGE_FEE);
81      } else {
82          // request token price
83          self.get_price_for_create_bot(&pair, &user, slippage, &entry_price_256, &mut
                new_grid_bot);
84      }
85  }
```

Listing 2.28: grid_bot.rs

```
13  pub fn deposit_near_to_get_wnear_for_create_bot(&mut self, pair: &Pair, user: &AccountId,
        slippage: u16, entry_price: &U256C,
14                              grid_bot: &mut GridBot, amount: u128) {
15      ext_wnear::ext(self.wnear.clone())
16          .with_attached_deposit(amount)
17          // .with_static_gas(GAS_FOR_CREATE_BOT_AFTER_NEAR)
18          .near_deposit()
19          .then(
20          Self::ext(env::current_account_id())
21              .with_static_gas(GAS_FOR_CREATE_BOT_AFTER_NEAR)
22              .after_wrap_near_for_create_bot(
23                  pair,
24                  user,
25                  slippage,
26                  entry_price,
27                  grid_bot,
28                  amount,
29              )
30      );
31  }
```

Listing 2.29: grid_bot.rs

```
29  pub fn deposit_near_to_get_wnear_for_create_bot(&mut self, pair: &Pair, user: &AccountId,
        slippage: u16, entry_price: &U256C,
30                              grid_bot: &mut GridBot, amount: u128) {
31      ext_wnear::ext(self.wnear.clone())
32          .with_attached_deposit(amount)
33          // .with_static_gas(GAS_FOR_CREATE_BOT_AFTER_NEAR)
34          .near_deposit()
35          .then(
36          Self::ext(env::current_account_id())
37              .with_static_gas(GAS_FOR_CREATE_BOT_AFTER_NEAR)
38              .after_wrap_near_for_create_bot(
39                  pair,
40                  user,
41                  slippage,
42                  entry_price,
43                  grid_bot,
44                  amount,
45              )
46      );
47  }
```

Listing 2.30: wnear.rs

```
62  fn after_wrap_near_for_create_bot(&mut self, pair: &Pair, user: &AccountId, slippage: u16,
        entry_price: &U256C, grid_bot: &mut GridBot, amount: u128) -> bool {
63      let promise_success = is_promise_success();
64      if !promise_success.clone() {
65          // refund token and near
66          self.internal_create_bot_refund_with_near(user, pair, amount + STORAGE_FEE,
                WRAP_TO_WNEAR_ERROR);
```

```
67          emit::wrap_near_error(user, 0, amount, true);
68      } else {
69          // deposit
70          if !self.internal_deposit(&user.clone(), &self.wnear.clone(), U128::from(amount)) {
71              // maybe just need hande one token, but it's ok, no problem
72              self.internal_create_bot_refund(user, pair, WRAP_TO_WNEAR_ERROR);
73              emit::wrap_near_error(user, 0, amount, true);
74          } else {
75              // request price
76              self.get_price_for_create_bot(pair, user, slippage, entry_price, grid_bot);
77          }
78      }
79      promise_success
80  }
```

Listing 2.31: wnear.rs

```
209 pub fn internal_deposit(&mut self, sender_id: &AccountId, token_in: &AccountId, amount: U128)
        -> bool {
210     require!(self.global_balances_map.contains_key(token_in), INVALID_TOKEN);
211     // require min deposit
212     // require!(amount.clone().0 >= self.deposit_limit_map.get(token_in).unwrap().as_u128(),
            LESS_DEPOSIT_AMOUNT);
213     if amount.clone().0 < self.deposit_limit_map.get(token_in).unwrap().as_u128() {
214         self.internal_token_refund(sender_id, token_in, LESS_DEPOSIT_AMOUNT);
215         emit::deposit_failed(sender_id, amount.clone().0, token_in);
216         return false;
217     }
218     // log!("Deposit user:{}, token:{}, amount:{}", sender_id.clone(), token_in.clone(), amount.
            clone().0);
219     // add amount to user
220     self.internal_increase_asset(sender_id, token_in, &(U256C::from(amount.clone().0)));
221     // add amount to global
222     self.internal_increase_global_asset(token_in, &(U256C::from(amount.clone().0)));
223     // event
224     emit::deposit_success(sender_id, amount.clone().0, token_in);
225     return true;
226 }
```

Listing 2.32: grid_bot_asset.rs

```
304 pub fn internal_create_bot_refund(&mut self, user: &AccountId, pair: &Pair, reason: &str) {
305     self.internal_withdraw_all(user, &pair.base_token);
306     self.internal_withdraw_all(user, &pair.quote_token);
307     emit::create_bot_error(user, reason);
308 }
```

Listing 2.33: grid_bot_asset.rs

Impact The refund balance of `WNEAR` is incorrect.

Suggestion Revise the logic accordingly.

2.1.11 Lack of Check in function close_bot()

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Function `close_bot()` allows the owner of a `grid_bot` to close it, decrease the locked balance, and retrieve the assets held within the contract. However, this function does not check whether the `grid_bot` has already been closed. In this case, malicious users can close the old closed `grid_bot` again after creating a new one. This can result in a created bot with zero locked balance and the whole state of the contract is incorrect. Furthermore, the normal take orders may not be taken, resulting in a Denial of Service.

```
88  pub fn close_bot(&mut self, bot_id: String) {
89      assert_one_yocto();
90      require!(self.bot_map.contains_key(&bot_id), BOT_NOT_EXIST);
91      let mut bot = self.bot_map.get(&bot_id).unwrap().clone();
92      let pair = self.pair_map.get(&bot.pair_id).unwrap().clone();
93      // check permission, user self close or take profit or stop loss
94      // let user = env::predecessor_account_id();
95      require!(env::predecessor_account_id() == bot.user, INVALID_USER);
96      // require!(self.internal_check_bot_close_permission(&user, &bot), NO_PERMISSION);
97
98
99      self.internal_close_bot(&env::predecessor_account_id(), &bot_id, &mut bot, &pair);
100 }
```

Listing 2.34: grid_bot.rs

```
102 pub fn internal_close_bot(&mut self, user: &AccountId, bot_id: &String, bot: &mut GridBot,
        pair: &Pair) {
103     // sign closed
104     bot.closed = true;
105
106
107     // harvest revenue, must fist execute, will split revenue from bot's asset
108     let (revenue_token, revenue) = self.internal_harvest_revenue(bot, pair);
109     // unlock token
110     self.internal_transfer_assets_to_unlock(&(bot.user), &(pair.base_token), bot.
            total_base_amount.clone());
111     self.internal_transfer_assets_to_unlock(&(bot.user), &(pair.quote_token), bot.
            total_quote_amount.clone());
112
113
114     // withdraw token
115     self.internal_withdraw(&(bot.user), &(pair.base_token), bot.total_base_amount);
116     self.internal_withdraw(&(bot.user), &(pair.quote_token), bot.total_quote_amount);
117     self.internal_withdraw(&(bot.user), &revenue_token, revenue);
118     self.bot_map.insert(bot_id, &bot);
119
120
121     // send claim event
```

```
122     if revenue.as_u128() > 0 {
123         // claim event
124         emit::claim(&user, &(bot.user), bot_id.clone(), &revenue_token, revenue);
125     }
126     emit::close_bot(user, bot_id.clone());
127 }
```

Listing 2.35: grid_bot_internal.rs

Impact The whole state of the contract can be wrong and the take orders cannot be taken.

Suggestion Add the check to ensure the `grid_bot` is not closed before closing it.

2.1.12 Lack of State Rollback in Callback Function

Severity High

Status Confirmed

Introduced by `Version 1`

Description Function `register_pair()` is designed to add new token pairs. Inside this function, `storage_deposit()` is invoked to deposit the storage fee for the registered token. However, in the callback function `after_storage_deposit()`, the state is not rolled back when the promise fails, which is incorrect.

```
308 pub fn internal_init_token(&mut self, token: AccountId, min_deposit: U128) -> U256C {
309     if self.global_balances_map.contains_key(&token) {
310         return U256C::from(0);
311     }
312     self.global_balances_map.insert(&token, &U256C::from(0));
313     self.protocol_fee_map.insert(&token, &U256C::from(0));
314     self.deposit_limit_map.insert(&token, &U256C::from(min_deposit.0));
315     self.internal_storage_deposit(&env::current_account_id(), &token, DEFAULT_TOKEN_STORAGE_FEE)
            ;
316     return U256C::from(DEFAULT_TOKEN_STORAGE_FEE);
317 }
```

Listing 2.36: grid_bot_internal.rs

```
39  pub fn internal_storage_deposit(&mut self, account_id: &AccountId, token_id: &AccountId,
        amount: Balance) -> Promise {
40      ext_fungible_token::ext(token_id.clone())
41          .with_attached_deposit(amount)
42          .with_static_gas(GAS_FOR_FT_TRANSFER)
43          .storage_deposit(
44              Some(account_id.clone()),
45              Some(true),
46          ).then(
47          Self::ext(env::current_account_id())
48              .with_static_gas(GAS_FOR_AFTER_FT_TRANSFER)
49              .after_storage_deposit(
50                  account_id.clone(),
51                  token_id.clone(),
52                  amount.into(),
```

```
53              )
54      )
55  }
```

Listing 2.37: token.rs

```
173 fn after_storage_deposit(
174     &mut self,
175     account_id: AccountId,
176     token_id: AccountId,
177     amount: U128,
178 ) -> bool {
179     let promise_success = is_promise_success();
180     if !promise_success.clone() {
181         emit::storage_deposit_failed(&account_id, amount.clone().0, &token_id);
182     } else {
183         emit::storage_deposit_succeeded(&account_id, amount.clone().0, &token_id);
184     }
185     promise_success
186 }
```

Listing 2.38: token.rs

Impact The contract state can be wrong.

Suggestion In the callback function `after_storage_deposit()`, implement related logic to rollback the status when the promise fails.

Feedback Rollback is not required here. If the invoke fails, it can be remedied by invoking the contract's function `storage_deposit()` again.

2.1.13 Redundant Refund Logic in Function internal_check_bot_amount()

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `create_bot()`, the function `internal_check_bot_amount()` is invoked to check if the user's input is valid, if not, the function `internal_create_bot_refund()` will be invoked to refund the user's assets and the returned result will be false. In this case, the function `internal_create_bot_refund_with_near()` will be invoked to refund the user's assets again. This redundant logic of withdrawal and refunding of user's assets is a waste of gas.

```
13      pub fn create_bot(&mut self, name: String, pair_id: String, slippage: u16, grid_type:
            GridType,
14                  grid_rate: u16, grid_offset: U128, first_base_amount: U128, first_quote_amount
                        : U128,
15                  last_base_amount: U128, last_quote_amount: U128, fill_base_or_quote: bool,
                        grid_sell_count: u16, grid_buy_count: u16,
16                  trigger_price: U128, take_profit_price: U128, stop_loss_price: U128,
                        valid_until_time: U128,
17                  entry_price: U128) {
18      let grid_offset_256 = U256C::from(grid_offset.0);
```

```
19      let first_base_amount_256 = U256C::from(first_base_amount.0);
20      let first_quote_amount_256 = U256C::from(first_quote_amount.0);
21      let last_base_amount_256 = U256C::from(last_base_amount.0);
22      let last_quote_amount_256 = U256C::from(last_quote_amount.0);
23      let trigger_price_256 = U256C::from(trigger_price.0);
24      let take_profit_price_256 = U256C::from(take_profit_price.0);
25      let stop_loss_price_256 = U256C::from(stop_loss_price.0);
26      let valid_until_time_256 = U256C::from(valid_until_time.0);
27      let entry_price_256 = U256C::from(entry_price.0);
28
29
30      require!(self.pair_map.contains_key(&pair_id), INVALID_PAIR_ID);
31      let pair = self.pair_map.get(&pair_id).unwrap().clone();
32      let user = env::predecessor_account_id();
33
34
35      // require!(self.status == GridStatus::Running, PAUSE_OR_SHUTDOWN);
36      if self.status != GridStatus::Running {
37          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
38          return;
39      }
40
41
42      if grid_buy_count + grid_sell_count > MAX_GRID_COUNT {
43          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
44          return;
45      }
46
47
48      // calculate all assets
49      let (base_amount_sell, quote_amount_buy) = GridBotContract::internal_calculate_bot_assets(
            first_quote_amount_256.clone(), last_base_amount_256.clone(), grid_sell_count.clone(),
             grid_buy_count.clone(),
50                                                  grid_type.clone(), grid_rate.clone(),
                                                        grid_offset_256.clone(), fill_base_or_quote.
                                                        clone());
51
52
53      // require!(env::attached_deposit() >= STORAGE_FEE, LESS_STORAGE_FEE);
54      if !self.internal_check_near_amount(&user, &pair, env::attached_deposit(), base_amount_sell
            , quote_amount_buy) {
55          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
56          return;
57      }
58      // last_quote_amount / last_base_amount > first_quote_amount > first_base_amount
59      // amount must u128, u128 * u128 <= u256, so, it's ok
60      let (result, reason) = self.internal_check_bot_amount(grid_sell_count, grid_buy_count,
            first_base_amount_256, first_quote_amount_256,
61                                                      last_base_amount_256, last_quote_amount_256,
                                                            &user, &pair, base_amount_sell,
```

```
                                                                quote_amount_buy);
62      if !result {
63          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(), &
                reason);
64          return;
65      }
66
67
68      // create bot
69      let mut new_grid_bot = GridBot {name, active: false, user: user.clone(), bot_id: "".
            to_string(), closed: false, pair_id, grid_type,
70          grid_sell_count: grid_sell_count.clone(), grid_buy_count: grid_buy_count.clone(),
                grid_rate, grid_offset: grid_offset_256,
71          first_base_amount: first_base_amount_256, first_quote_amount: first_quote_amount_256,
                last_base_amount: last_base_amount_256,
72          last_quote_amount: last_quote_amount_256, fill_base_or_quote, trigger_price:
                trigger_price_256, trigger_price_above_or_below: false,
73          take_profit_price: take_profit_price_256, stop_loss_price: stop_loss_price_256,
                valid_until_time: valid_until_time_256,
74          total_quote_amount: quote_amount_buy, total_base_amount: base_amount_sell, revenue:
                U256C::from(0), total_revenue: U256C::from(0)
75      };
76
77
78      if self.internal_need_wrap_near(&user, &pair, base_amount_sell, quote_amount_buy) {
79          // wrap near to wnear first
80          self.deposit_near_to_get_wnear_for_create_bot(&pair, &user, slippage, &entry_price_256,
                &mut new_grid_bot, env::attached_deposit() - STORAGE_FEE);
81      } else {
82          // request token price
83          self.get_price_for_create_bot(&pair, &user, slippage, &entry_price_256, &mut
                new_grid_bot);
84      }
85  }
```

Listing 2.39: gird_bot.rs

```
25      pub fn internal_check_bot_amount(&mut self, grid_sell_count: u16, grid_buy_count: u16,
            first_base_amount_256: U256C, first_quote_amount_256: U256C,
26                                  last_base_amount_256: U256C, last_quote_amount_256: U256C, user:
                                        &AccountId, pair: &Pair, base_amount_sell: U256C,
                                        quote_amount_buy: U256C) -> (bool, String) {
27      if grid_sell_count > 0 && grid_buy_count > 0 {
28          // require!(last_quote_amount_256 * first_base_amount_256 > first_quote_amount_256 *
                last_base_amount_256 , INVALID_FIRST_OR_LAST_AMOUNT);
29          if last_quote_amount_256 * first_base_amount_256 <= first_quote_amount_256 *
                last_base_amount_256 {
30              return (false, INVALID_FIRST_OR_LAST_AMOUNT.to_string());
31          }
32      }
33      if grid_sell_count > 0 {
34          // require!(first_base_amount_256.as_u128() > 0 && first_quote_amount_256.as_u128() >
                0, INVALID_FIRST_OR_LAST_AMOUNT);
```

```
35          // if first_base_amount_256.as_u128() == 0 || first_quote_amount_256.as_u128() == 0 {
36          if last_base_amount_256.as_u128() == 0 || last_quote_amount_256.as_u128() == 0 {
37              return (false, INVALID_FIRST_OR_LAST_AMOUNT.to_string());
38          }
39          // require!(base_amount_sell.as_u128() / grid_sell_count as u128 >= self.
                deposit_limit_map.get(&pair.base_token).unwrap().as_u128(), BASE_TO_SMALL);
40          if (base_amount_sell.as_u128() / grid_sell_count as u128) < self.deposit_limit_map.get
                (&pair.base_token).unwrap().as_u128() {
41              return (false, BASE_TOO_SMALL.to_string());
42          }
43      }
44      if grid_buy_count > 0 {
45          // require!(last_base_amount_256.as_u128() > 0 && last_quote_amount_256.as_u128() > 0,
                INVALID_FIRST_OR_LAST_AMOUNT);
46          // if last_base_amount_256.as_u128() == 0 || last_quote_amount_256.as_u128() == 0 {
47          if first_base_amount_256.as_u128() == 0 || first_quote_amount_256.as_u128() == 0 {
48              return (false, INVALID_FIRST_OR_LAST_AMOUNT.to_string());
49          }
50          // require!(quote_amount_buy.as_u128() / grid_buy_count as u128 >= self.
                deposit_limit_map.get(&pair.quote_token).unwrap().as_u128(), QUOTE_TO_SMALL);
51          if (quote_amount_buy.as_u128() / grid_buy_count as u128) < self.deposit_limit_map.get(&
                pair.quote_token).unwrap().as_u128() {
52              self.internal_create_bot_refund(&user, &pair, QUOTE_TOO_SMALL);
53              return (false, QUOTE_TOO_SMALL.to_string());
54          }
55      }
56      return (true, "".to_string());
57  }
```

Listing 2.40: gird_bot_check.rs

```
299  pub fn internal_create_bot_refund_with_near(&mut self, user: &AccountId, pair: &Pair,
         near_amount: u128, reason: &str) {
300     self.internal_create_bot_refund(user, pair, reason);
301     self.internal_near_refund(user, near_amount);
302  }
```

Listing 2.41: gird_bot_asset.rs

```
304  pub fn internal_create_bot_refund(&mut self, user: &AccountId, pair: &Pair, reason: &str) {
305      self.internal_withdraw_all(user, &pair.base_token);
306      self.internal_withdraw_all(user, &pair.quote_token);
307      emit::create_bot_error(user, reason);
308  }
```

Listing 2.42: gird_bot_asset.rs

Impact Redundant refund logic can lead to gas waste.

Suggestion Remove the redundant refund operation in the function `internal_check_bot_amount()`.

2.1.14 Lack of Proper Handling of Token Decimals

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the current implementation, the decimals of tokens are not scaled. When a user creates a `grid_bot`, the quantities of the two tokens they provide may differ significantly due to differences in their decimal places. In such cases, when calculating the matched amount of tokens in the function `internal_check_order_match()`, there is a possibility of encountering precision loss, which can lead to calculation errors.

```
97   pub fn internal_check_order_match(maker_order: Order, taker_order: Order) {
98     require!(maker_order.token_buy == taker_order.token_sell, INVALID_ORDER_TOKEN);
99     require!(maker_order.token_sell == taker_order.token_buy, INVALID_ORDER_TOKEN);
100    require!(taker_order.token_sell != taker_order.token_buy, INVALID_ORDER_TOKEN);
101    // taker price and maker price match
102    require!(BigDecimal::from(taker_order.amount_sell.as_u128()).div(BigDecimal::from(
           taker_order.amount_buy.as_u128())) >= BigDecimal::from(maker_order.amount_buy.as_u128()
           ).div(BigDecimal::from(maker_order.amount_sell.as_u128())), INVALID_PRICE);
103  }
```

Listing 2.43: orderbook_internal.rs

Impact Calculation errors caused by precision loss can potentially prevent orders from being matched properly.

Suggestion When performing calculations, it is recommended to configure a scaled decimal for different tokens.

2.1.15 Gas Waste due to Redundant Checks in Function internal_create_bot()

Severity Low

Status Confirmed

Introduced by `Version 1`

Description Function `create_bot()` invokes function `internal_create_bot()` to create a `grid_bot`. Both functions check if the `DeltaBot`'s status is `Running`, resulting in wasted gas due to the duplication of this check.

```
13     pub fn create_bot(&mut self, name: String, pair_id: String, slippage: u16, grid_type:
           GridType,
14                 grid_rate: u16, grid_offset: U128, first_base_amount: U128, first_quote_amount
                       : U128,
15                 last_base_amount: U128, last_quote_amount: U128, fill_base_or_quote: bool,
                       grid_sell_count: u16, grid_buy_count: u16,
16                 trigger_price: U128, take_profit_price: U128, stop_loss_price: U128,
                       valid_until_time: U128,
17                 entry_price: U128) {
18     let grid_offset_256 = U256C::from(grid_offset.0);
19     let first_base_amount_256 = U256C::from(first_base_amount.0);
20     let first_quote_amount_256 = U256C::from(first_quote_amount.0);
```

```
21      let last_base_amount_256 = U256C::from(last_base_amount.0);
22      let last_quote_amount_256 = U256C::from(last_quote_amount.0);
23      let trigger_price_256 = U256C::from(trigger_price.0);
24      let take_profit_price_256 = U256C::from(take_profit_price.0);
25      let stop_loss_price_256 = U256C::from(stop_loss_price.0);
26      let valid_until_time_256 = U256C::from(valid_until_time.0);
27      let entry_price_256 = U256C::from(entry_price.0);
28
29
30      require!(self.pair_map.contains_key(&pair_id), INVALID_PAIR_ID);
31      let pair = self.pair_map.get(&pair_id).unwrap().clone();
32      let user = env::predecessor_account_id();
33
34
35      // require!(self.status == GridStatus::Running, PAUSE_OR_SHUTDOWN);
36      if self.status != GridStatus::Running {
37          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
38          return;
39      }
40
41
42      if grid_buy_count + grid_sell_count > MAX_GRID_COUNT {
43          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
44          return;
45      }
46
47
48      // calculate all assets
49      let (base_amount_sell, quote_amount_buy) = GridBotContract::internal_calculate_bot_assets(
            first_quote_amount_256.clone(), last_base_amount_256.clone(), grid_sell_count.clone(),
             grid_buy_count.clone(),
50                                                  grid_type.clone(), grid_rate.clone(),
                                                        grid_offset_256.clone(), fill_base_or_quote.
                                                        clone());
51
52
53      // require!(env::attached_deposit() >= STORAGE_FEE, LESS_STORAGE_FEE);
54      if !self.internal_check_near_amount(&user, &pair, env::attached_deposit(), base_amount_sell
            , quote_amount_buy) {
55          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
56          return;
57      }
58      // last_quote_amount / last_base_amount > first_quote_amount > first_base_amount
59      // amount must u128, u128 * u128 <= u256, so, it's ok
60      let (result, reason) = self.internal_check_bot_amount(grid_sell_count, grid_buy_count,
            first_base_amount_256, first_quote_amount_256,
61                                                            last_base_amount_256, last_quote_amount_256,
                                                                 &user, &pair, base_amount_sell,
                                                                quote_amount_buy);
62      if !result {
```

```
63          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(), &
                reason);
64          return;
65      }
66
67
68      // create bot
69      let mut new_grid_bot = GridBot {name, active: false, user: user.clone(), bot_id: "".
            to_string(), closed: false, pair_id, grid_type,
70          grid_sell_count: grid_sell_count.clone(), grid_buy_count: grid_buy_count.clone(),
                grid_rate, grid_offset: grid_offset_256,
71          first_base_amount: first_base_amount_256, first_quote_amount: first_quote_amount_256,
                last_base_amount: last_base_amount_256,
72          last_quote_amount: last_quote_amount_256, fill_base_or_quote, trigger_price:
                trigger_price_256, trigger_price_above_or_below: false,
73          take_profit_price: take_profit_price_256, stop_loss_price: stop_loss_price_256,
                valid_until_time: valid_until_time_256,
74          total_quote_amount: quote_amount_buy, total_base_amount: base_amount_sell, revenue:
                U256C::from(0), total_revenue: U256C::from(0)
75      };
76
77
78      if self.internal_need_wrap_near(&user, &pair, base_amount_sell, quote_amount_buy) {
79          // wrap near to wnear first
80          self.deposit_near_to_get_wnear_for_create_bot(&pair, &user, slippage, &entry_price_256,
                &mut new_grid_bot, env::attached_deposit() - STORAGE_FEE);
81      } else {
82          // request token price
83          self.get_price_for_create_bot(&pair, &user, slippage, &entry_price_256, &mut
                new_grid_bot);
84      }
85  }
```

Listing 2.44: gird_bot.rs

```
15      pub fn internal_create_bot(&mut self,
16                             base_price: Price,
17                             quote_price: Price,
18                             user: &AccountId,
19                             slippage: u16,
20                             entry_price: &U256C,
21                             pair: &Pair,
22                             grid_bot: &mut GridBot) -> bool {
23      if self.status != GridStatus::Running {
24          self.internal_create_bot_refund(&user, &pair, PAUSE_OR_SHUTDOWN);
25          return false;
26      }
27      // require!(self.internal_check_oracle_price(*entry_price, base_price.clone(), quote_price.
            clone(), slippage), INVALID_PRICE);
28      if !self.internal_check_oracle_price(*entry_price, base_price.clone(), quote_price.clone(),
             slippage) {
29          self.internal_create_bot_refund(user, pair, INVALID_PRICE);
30          return false;
```

```
31      }
32      // check balance
33      // require!(self.internal_get_user_balance(user, &(pair.base_token)) >= base_amount_sell,
            LESS_BASE_TOKEN);
34      if self.internal_get_user_balance(user, &(pair.base_token)) < grid_bot.total_base_amount {
35          self.internal_create_bot_refund(user, pair, LESS_BASE_TOKEN);
36          return false;
37      }
38      // require!(self.internal_get_user_balance(user, &(pair.quote_token)) >= quote_amount_buy,
            LESS_QUOTE_TOKEN);
39      if self.internal_get_user_balance(user, &(pair.quote_token)) < grid_bot.total_quote_amount
            {
40          self.internal_create_bot_refund(user, pair, LESS_QUOTE_TOKEN);
41          return false;
42      }
43
44
45      // create bot id
46      let next_bot_id = format!("GRID:{}", self.internal_get_and_use_next_bot_id().to_string());
47      grid_bot.bot_id = next_bot_id;
48
49
50      // initial orders space, create empty orders
51      let grid_count = grid_bot.grid_sell_count.clone() + grid_bot.grid_buy_count.clone();
52      self.create_default_orders(grid_bot.bot_id.clone(), grid_count);
53
54
55      // transfer assets
56      self.internal_transfer_assets_to_lock(&user, &pair.base_token, grid_bot.total_base_amount);
57      self.internal_transfer_assets_to_lock(&user, &pair.quote_token, grid_bot.total_quote_amount
            );
58
59
60      // init active status of bot
61      self.internal_init_bot_status(grid_bot, entry_price);
62
63
64      // insert bot
65      self.bot_map.insert(&(grid_bot.bot_id), &grid_bot);
66
67
68      emit::create_bot(&grid_bot.user, grid_bot.bot_id.clone(), base_price.price.0.to_string(),
            quote_price.price.0.to_string(), base_price.expo.to_string(), quote_price.expo.
            to_string());
69      return true;
70  }
```

Listing 2.45: grid_bot_internal.rs

Impact Redundant checks lead to waste of gas.

Suggestion Remove the check for the `DeltaBot` contract's status within the function `create_bot()`.

Feedback Since the process of creating a `grid_bot` involves cross-contract calls, secondary

verification is necessary to ensure that the contract is `Running`.

2.1.16 Unreasonable Logic in Function internal_check_near_amount()

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description In function `create_bot()`, the function `internal_check_near_amount()` is invoked to check whether the amount of the attached `NEAR` is correct. Specifically, when the user's balance of `WNEAR` is not enough, the contract requires the user to deposit not just the remaining amount needed, but rather the entire amount of `NEAR` required to create the `grid_bot`, which is unreasonable.

```
13  pub fn create_bot(&mut self, name: String, pair_id: String, slippage: u16, grid_type: GridType
        ,
14                  grid_rate: u16, grid_offset: U128, first_base_amount: U128, first_quote_amount
                        : U128,
15                  last_base_amount: U128, last_quote_amount: U128, fill_base_or_quote: bool,
                        grid_sell_count: u16, grid_buy_count: u16,
16                  trigger_price: U128, take_profit_price: U128, stop_loss_price: U128,
                        valid_until_time: U128,
17                  entry_price: U128) {
18      let grid_offset_256 = U256C::from(grid_offset.0);
19      let first_base_amount_256 = U256C::from(first_base_amount.0);
20      let first_quote_amount_256 = U256C::from(first_quote_amount.0);
21      let last_base_amount_256 = U256C::from(last_base_amount.0);
22      let last_quote_amount_256 = U256C::from(last_quote_amount.0);
23      let trigger_price_256 = U256C::from(trigger_price.0);
24      let take_profit_price_256 = U256C::from(take_profit_price.0);
25      let stop_loss_price_256 = U256C::from(stop_loss_price.0);
26      let valid_until_time_256 = U256C::from(valid_until_time.0);
27      let entry_price_256 = U256C::from(entry_price.0);
28
29
30      require!(self.pair_map.contains_key(&pair_id), INVALID_PAIR_ID);
31      let pair = self.pair_map.get(&pair_id).unwrap().clone();
32      let user = env::predecessor_account_id();
33
34
35      // require!(self.status == GridStatus::Running, PAUSE_OR_SHUTDOWN);
36      if self.status != GridStatus::Running {
37          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
38          return;
39      }
40
41
42      if grid_buy_count + grid_sell_count > MAX_GRID_COUNT {
43          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
44          return;
```

```
45      }
46
47
48      // calculate all assets
49      let (base_amount_sell, quote_amount_buy) = GridBotContract::internal_calculate_bot_assets(
            first_quote_amount_256.clone(), last_base_amount_256.clone(), grid_sell_count.clone(),
             grid_buy_count.clone(),
50                                                  grid_type.clone(), grid_rate.clone(),
                                                        grid_offset_256.clone(), fill_base_or_quote.
                                                        clone());
51
52
53      // require!(env::attached_deposit() >= STORAGE_FEE, LESS_STORAGE_FEE);
54      if !self.internal_check_near_amount(&user, &pair, env::attached_deposit(), base_amount_sell
            , quote_amount_buy) {
55          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
56          return;
57      }
58      // last_quote_amount / last_base_amount > first_quote_amount > first_base_amount
59      // amount must u128, u128 * u128 <= u256, so, it's ok
60      let (result, reason) = self.internal_check_bot_amount(grid_sell_count, grid_buy_count,
            first_base_amount_256, first_quote_amount_256,
61                                                            last_base_amount_256, last_quote_amount_256,
                                                                 &user, &pair, base_amount_sell,
                                                                quote_amount_buy);
62      if !result {
63          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(), &
                reason);
64          return;
65      }
66
67
68      // create bot
69      let mut new_grid_bot = GridBot {name, active: false, user: user.clone(), bot_id: "".
            to_string(), closed: false, pair_id, grid_type,
70          grid_sell_count: grid_sell_count.clone(), grid_buy_count: grid_buy_count.clone(),
                grid_rate, grid_offset: grid_offset_256,
71          first_base_amount: first_base_amount_256, first_quote_amount: first_quote_amount_256,
                last_base_amount: last_base_amount_256,
72          last_quote_amount: last_quote_amount_256, fill_base_or_quote, trigger_price:
                trigger_price_256, trigger_price_above_or_below: false,
73          take_profit_price: take_profit_price_256, stop_loss_price: stop_loss_price_256,
                valid_until_time: valid_until_time_256,
74          total_quote_amount: quote_amount_buy, total_base_amount: base_amount_sell, revenue:
                U256C::from(0), total_revenue: U256C::from(0)
75      };
76
77
78      if self.internal_need_wrap_near(&user, &pair, base_amount_sell, quote_amount_buy) {
79          // wrap near to wnear first
80          self.deposit_near_to_get_wnear_for_create_bot(&pair, &user, slippage, &entry_price_256,
                &mut new_grid_bot, env::attached_deposit() - STORAGE_FEE);
```

```
81      } else {
82          // request token price
83          self.get_price_for_create_bot(&pair, &user, slippage, &entry_price_256, &mut
                new_grid_bot);
84      }
85  }
```

Listing 2.46: grid_bot.rs

```
79  pub fn internal_check_near_amount(&mut self, user: &AccountId, pair: &Pair, near_amount: u128,
        base_amount_sell: U256C, quote_amount_buy: U256C) -> bool {
80      if pair.quote_token != self.wnear && pair.base_token != self.wnear && near_amount !=
            STORAGE_FEE {
81          return false;
82      }
83      let wnear_balance = self.internal_get_user_balance(&user, &self.wnear);
84      if pair.base_token == self.wnear {
85          if wnear_balance.as_u128() >= base_amount_sell.as_u128() && near_amount != STORAGE_FEE
                {
86              // wnear balance is enough, but user support near
87              return false;
88          }
89          if wnear_balance.as_u128() < base_amount_sell.as_u128() && near_amount != (
                base_amount_sell.as_u128() + STORAGE_FEE) { //audit
90              // wnear balance is not enough, but near is less
91              return false;
92          }
93      }
94      if pair.quote_token == self.wnear {
95          if wnear_balance.as_u128() >= quote_amount_buy.as_u128() && near_amount != STORAGE_FEE
                {
96              // wnear balance is enough, but user support near
97              return false;
98          }
99          if wnear_balance.as_u128() < quote_amount_buy.as_u128() && near_amount != (
                quote_amount_buy.as_u128() + STORAGE_FEE) {
100             // wnear balance is not enough, but near is less
101             return false;
102         }
103     }
104     // if wnear not register, will revert, it's ok
105     let wnear_min_deposit = self.deposit_limit_map.get(&self.wnear).unwrap();
106     if pair.base_token == self.wnear && base_amount_sell.as_u128() < wnear_min_deposit.as_u128
            ()
107         || pair.quote_token == self.wnear && quote_amount_buy.as_u128() < wnear_min_deposit.
                as_u128() {
108         return false;
109     }
110     return true;
111 }
```

Listing 2.47: grid_bot_check.rs

Impact If the user's balance of `WNEAR` is not sufficient to create the `grid_bot`, the user has to deposit the entire required amount of `NEAR` for creating that `grid_bot`, which is unreasonable.

Suggestion Implement the correct checking logic accordingly.

Feedback The `WNEAR` required for users to create a `grid_bot` is entirely provided by the `NEAR` attached to this invocation. If there is `WNEAR` in the user's `user_balance`, it must be withdrawn through the function `withdraw()`.

2.1.17 Incorrect Revenue Token Returned in Forward Order

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Function `internal_calculate_bot_revenue()` determines if the user is profitable based on whether their executed order is a `reverse` order. If not, the user did not make a profit, and the function returns the profit token as `token_sell` of opposite order, which is incorrect.

```
32  pub fn internal_take_order(&mut self, bot_id: String, forward_or_reverse: bool, level: usize,
        taker_order: &Order, took_sell: U256C, took_buy: U256C) -> (U256C, U256C, AccountId, U256C
        , U256C, U256C, U256C) {
33      let bot = self.bot_map.get(&bot_id.clone()).unwrap().clone();
34      let pair = self.pair_map.get(&bot.pair_id).unwrap().clone();
35      let (maker_order, in_orderbook) = self.query_order(bot_id.clone(), forward_or_reverse, level
            );
36      // matching check
37      GridBotContract::internal_check_order_match(maker_order.clone(), taker_order.clone());
38
39
40      // calculate
41      let (taker_sell, taker_buy, current_filled, made_order) = GridBotContract::
            internal_calculate_matching(maker_order.clone(), taker_order.clone(), took_sell,
            took_buy);
42
43
44      // place into orderbook
45      if !in_orderbook {
46          self.internal_place_order(bot_id.clone(), maker_order.clone(), forward_or_reverse.clone
                (), level.clone());
47      }
48      // update filled
49      let maker_order = self.internal_update_order_filled(bot_id.clone(), forward_or_reverse.clone
            (), level.clone(), current_filled.clone());
50      emit::order_update(bot_id.clone(), forward_or_reverse.clone(), level.clone(), &maker_order);
51
52
53      // place opposite order
54      let opposite_order = GridBotContract::internal_get_opposite_order(&made_order, bot.clone(),
            forward_or_reverse.clone(), level.clone());
55      self.internal_place_order(bot_id.clone(), opposite_order.clone(), !forward_or_reverse.clone
            (), level.clone());
56
```

```
57
58      // query real_opposite_order
59      let (real_opposite_order, _) = self.query_order(bot_id.clone(), !forward_or_reverse.clone(),
            level.clone());
60      emit::order_update(bot_id.clone(), !forward_or_reverse.clone(), level.clone(), &
            real_opposite_order);
61
62
63      // calculate bot's revenue
64      let (revenue_token, revenue, maker_fee) = self.internal_calculate_bot_revenue(
            forward_or_reverse.clone(), maker_order.clone(), opposite_order, current_filled.clone()
            );
65
66
67      // add revenue
68      // let bot_mut = self.bot_map.get_mut(&bot_id.clone()).unwrap();
69      let mut bot = self.bot_map.get(&bot_id.clone()).unwrap();
70      bot.revenue += revenue;
71      bot.total_revenue += revenue;
72      // update bot asset
73      GridBotContract::internal_update_bot_asset(&mut bot, &pair, taker_order.token_buy.clone(),
            taker_buy.as_u128(), taker_sell.as_u128());
74
75
76      // bot asset transfer
77      self.internal_reduce_locked_assets(&(bot.user), &(taker_order.token_buy), &taker_buy);
78      self.internal_increase_locked_assets(&(bot.user), &(taker_order.token_sell), &taker_sell);
79
80
81      // allocate refer fee
82      let (protocol_fee, _) = self.internal_allocate_refer_fee(&maker_fee, &bot.user, &
            revenue_token);
83      // handle protocol fee
84      self.internal_add_protocol_fee_from_revenue(&mut bot, &revenue_token, maker_fee,
            protocol_fee, &pair);
85
86
87      // update bot
88      self.bot_map.insert(&bot_id, &bot);
89
90
91      // log!("Success take order, maker bot id:{}, forward_or_reserve:{}, level:{}, took sell:{},
            took buy:{}", bot_id, forward_or_reverse, level, taker_sell, taker_buy);
92      return (taker_sell, taker_buy, bot.user.clone(), maker_fee, revenue, bot.revenue, bot.
            total_revenue);
93  }
```

Listing 2.48: orderbook_internal.rs

```
224  pub fn internal_calculate_bot_revenue(&self, forward_or_reverse: bool, order: Order,
         opposite_order: Order, current_filled: U256C) -> (AccountId, U256C, U256C) {
225      if forward_or_reverse {
226          return (opposite_order.token_sell, U256C::from(0), U256C::from(0));
```

```
227         }
228         // let forward_order = GridBotContract::internal_get_first_forward_order(bot, pair, level);
229         let revenue_token;
230         let mut revenue;
231         // TODO had made_order, maybe can use mad_order
232         // mad_order, opposite_order
233         if opposite_order.fill_buy_or_sell {
234             // current_filled token is forward_order's buy token
235             // revenue token is forward_order's sell token
236             let forward_sold = current_filled.clone() * opposite_order.amount_sell / opposite_order
                    .amount_buy;
237             let reverse_bought = current_filled.clone() * order.amount_buy / order.amount_sell;
238             require!(reverse_bought >= forward_sold, INVALID_REVENUE);
239             revenue_token = opposite_order.token_sell;
240             revenue = reverse_bought - forward_sold;
241         } else {
242             // current_filled token is forward_order's sell token
243             // revenue token is forward_order's buy token
244             let forward_bought = current_filled.clone() * opposite_order.amount_buy /
                    opposite_order.amount_sell;
245             let reverse_sold = current_filled.clone() * order.amount_sell / order.amount_buy;
246             require!(forward_bought >= reverse_sold, INVALID_REVENUE);
247             revenue_token = opposite_order.token_buy;
248             revenue = forward_bought - reverse_sold;
249         };
250         let protocol_fee = revenue * U256C::from(self.protocol_fee_rate.clone()) / U256C::from(
                PROTOCOL_FEE_DENOMINATOR);
251         revenue -= protocol_fee;
252         return (revenue_token, revenue.clone(), protocol_fee.clone());
253     }
254
255
256     pub fn internal_calculate_taker_fee(&self, took_buy: U256C) -> (U256C, U256C) {
257         let taker_fee = took_buy * U256C::from(self.taker_fee_rate.clone()) / U256C::from(
                PROTOCOL_FEE_DENOMINATOR);
258         return (took_buy - taker_fee, taker_fee);
259     }
```

Listing 2.49: orderbook_internal.rs

Impact The returned revenue token is not correct, which is against the design.

Suggestion The revenue token should be the opposite order's `token_buy`.

2.1.18 Function token_storage_deposit() Fails to Deposit Storage Fees

Severity High

Status Fixed in `Version 3`

Introduced by `Version 2`

Description Any user can invoke the function `token_storage_deposit()` to pay the storage fees of their token balances. This function is designed for the payment of storage fees and

does not involve the actual deposit of any tokens. Consequently, the `amount` parameter passed to the functions `internal_increase_asset()` and `internal_increase_locked_assets()` is zero. However, due to the condition on line 57 in the function `internal_increase_locked_assets()`, users are unable to pay storage fees for their locked balances, which is incorrect.

```
167   #[payable]
168   pub fn token_storage_deposit(&mut self, user: AccountId, token: AccountId) {
169       require!(env::attached_deposit() == BASE_CREATE_STORAGE_FEE);
170       let initial_storage_usage = env::storage_usage();
171       self.internal_increase_asset(&user, &token, &U256C::from(0));
172       self.internal_increase_locked_assets(&user, &token, &U256C::from(0));
173       self.internal_refund_deposit(BASE_CREATE_STORAGE_FEE, initial_storage_usage, &env::
              predecessor_account_id());
174   }
```

Listing 2.50: grid_bot.rs

```
56   pub fn internal_increase_locked_assets(&mut self, user: &AccountId, token: &AccountId, amount:
         &U256C) {
57       if *amount == U256C::from(0) {
58           return;
59       }
60       let mut user_locked_balances = self.user_locked_balances_map.get(user).unwrap_or_else(|| {
61           let mut map = LookupMap::new(StorageKey::UserLockedBalanceSubKey(user.clone()));
62           map.insert(token, &U256C::from(0));
63           map
64       });
65
66
67       let balance = user_locked_balances.get(token).unwrap_or(U256C::from(0));
68       user_locked_balances.insert(token, &(balance + amount));
69
70
71       self.user_locked_balances_map.insert(user, &user_locked_balances);
72   }
```

Listing 2.51: grid_bot_asset.rs

Impact The function `token_storage_deposit()` fails to properly deposit storage fees for users.

Suggestion Modify the conditional logic in the function `internal_increase_locked_assets()` to ensure correct deposit of storage fees.

2.1.19 Lack of Check on Parameter in Function token_storage_deposit()

Severity Low

Status Fixed in `Version 3`

Introduced by `Version 2`

Description Function `token_storage_deposit()` does not check if the token is in the whitelist, allowing users to deposit storage fees for any token. However, when users attempt to deposit the corresponding token, function `internal_deposit()` verifies whether the token is in

the whitelist. Consequently, for the same token, users can deposit storage fees but cannot deposit the token itself. This inconsistency is incorrect.

```
167   #[payable]
168   pub fn token_storage_deposit(&mut self, user: AccountId, token: AccountId) {
169       require!(env::attached_deposit() == BASE_CREATE_STORAGE_FEE);
170       let initial_storage_usage = env::storage_usage();
171       self.internal_increase_asset(&user, &token, &U256C::from(0));
172       self.internal_increase_locked_assets(&user, &token, &U256C::from(0));
173       self.internal_refund_deposit(BASE_CREATE_STORAGE_FEE, initial_storage_usage, &env::
              predecessor_account_id());
174   }
```

Listing 2.52: grid_bot.rs

```
18   ft_on_transfer(
19       &mut self,
20       sender_id: AccountId,
21       amount: U128,
22       msg: String,
23   ) -> PromiseOrValue<U128> {
24       let token_in = env::predecessor_account_id();
25       if msg.is_empty() {
26           if !self.internal_deposit(&sender_id, &token_in, amount) {
27               return PromiseOrValue::Value(amount);
28           } else {
29               return PromiseOrValue::Value(U128::from(0));
30           }
31       } else {
32           let left = self.internal_parse_take_request(&sender_id, &token_in, amount, msg);
33           return PromiseOrValue::Value(left);
34       }
35   }
```

Listing 2.53: token.rs

```
163   pub fn internal_deposit(&mut self, sender_id: &AccountId, token_in: &AccountId, amount: U128)
          -> bool {
164       require!(self.global_balances_map.contains_key(token_in), INVALID_TOKEN);
165       if !self.query_user_token_registered(sender_id.clone(), token_in.clone()) {
166           emit::deposit_failed(sender_id, amount.clone().0, token_in);
167           return false;
168       }
169       // require min deposit
170       if amount.clone().0 < self.deposit_limit_map.get(token_in).unwrap().as_u128() {
171           emit::deposit_failed(sender_id, amount.clone().0, token_in);
172           return false;
173       }
174       // log!("Deposit user:{}, token:{}, amount:{}", sender_id.clone(), token_in.clone(), amount
              .clone().0);
175       // add amount to user
176       self.internal_increase_asset(sender_id, token_in, &(U256C::from(amount.clone().0)));
177       // add amount to global
178       self.internal_increase_global_asset(token_in, &(U256C::from(amount.clone().0)));
```

```
179         // event
180         emit::deposit_success(sender_id, amount.clone().0, token_in);
181         return true;
182     }
```

Listing 2.54: grid_bot_asset.rs

Impact Users may lose the corresponding storage fees.

Suggestion Add a check to ensure that the token passed into the function `token_storage_deposit()` is in the contract's whitelist.

2.1.20 Incorrect Storage_Key in Function internal_add_refer_recommend_user()

Severity High

Status Fixed in `Version 3`

Introduced by `Version 2`

Description Function `internal_add_referral_user()` records the relationship between the user and recommender. However, in the function `internal_add_refer_recommend_user()`, the user's accountId is used as the `Storage_Key`, which does not guarantee uniqueness. When the same user creates two grid_bots with different recommenders, the value of `refer_recommender_user_map` for different recommenders is recorded with the same `Storage_Key`. Consequently, the second entry will overwrite the first.

```
13    pub fn create_bot(&mut self, name: String, pair_id: String, slippage: u16, grid_type: GridType
          ,
14                  grid_rate: u16, grid_offset: U128, first_base_amount: U128, first_quote_amount:
                        U128,
15                  last_base_amount: U128, last_quote_amount: U128, fill_base_or_quote: bool,
                        grid_sell_count: u16, grid_buy_count: u16,
16                  trigger_price: U128, take_profit_price: U128, stop_loss_price: U128,
                        valid_until_time: U128,
17                  entry_price: U128, recommender: Option<AccountId>) {
18        // record storage fee
19        let initial_storage_usage = env::storage_usage();
20        let grid_offset_256 = U256C::from(grid_offset.0);
21        let first_base_amount_256 = U256C::from(first_base_amount.0);
22        let first_quote_amount_256 = U256C::from(first_quote_amount.0);
23        let last_base_amount_256 = U256C::from(last_base_amount.0);
24        let last_quote_amount_256 = U256C::from(last_quote_amount.0);
25        let trigger_price_256 = U256C::from(trigger_price.0);
26        let take_profit_price_256 = U256C::from(take_profit_price.0);
27        let stop_loss_price_256 = U256C::from(stop_loss_price.0);
28        let valid_until_time_256 = U256C::from(valid_until_time.0);
29        let entry_price_256 = U256C::from(entry_price.0);
30
31
32        require!(valid_until_time.0 > env::block_timestamp_ms() as u128, INVALID_UNTIL_TIME);
33
34
35        require!(self.pair_map.contains_key(&pair_id), INVALID_PAIR_ID);
```

```
36      let pair = self.pair_map.get(&pair_id).unwrap().clone();
37      let user = env::predecessor_account_id();
38
39
40      // require!(self.status == GridStatus::Running, PAUSE_OR_SHUTDOWN);
41      if self.status != GridStatus::Running {
42          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
43          return;
44      }
45
46
47      if grid_buy_count + grid_sell_count > MAX_GRID_COUNT {
48          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                MORE_THAN_MAX_GRID_COUNT);
49          return;
50      }
51
52
53      // calculate all assets
54      let (base_amount_sell, quote_amount_buy) = GridBotContract::internal_calculate_bot_assets(
            first_quote_amount_256.clone(), last_base_amount_256.clone(), grid_sell_count.clone(),
55                grid_buy_count.clone(),grid_type.clone(), grid_rate.clone(), grid_offset_256.
                      clone(), fill_base_or_quote.clone());
56
57
58      // require!(env::attached_deposit() >= STORAGE_FEE, LESS_STORAGE_FEE);
59      if !self.internal_check_near_amount(&user, &pair, env::attached_deposit(), base_amount_sell
            , quote_amount_buy) {
60          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                INVALID_AMOUNT);
61          return;
62      }
63      // last_quote_amount / last_base_amount > first_quote_amount > first_base_amount
64      // amount must u128, u128 * u128 <= u256, so, it's ok
65      let (result, reason) = self.internal_check_bot_amount(grid_sell_count, grid_buy_count,
            first_base_amount_256, first_quote_amount_256,
66                                          last_base_amount_256, last_quote_amount_256, &pair,
                                                base_amount_sell, quote_amount_buy);
67      if !result {
68          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(), &
                reason);
69          return;
70      }
71
72
73      // create bot
74      let mut new_grid_bot = GridBot {name, active: false, user: user.clone(), bot_id: "".
            to_string(), closed: false, pair_id, grid_type,
75          grid_sell_count: grid_sell_count.clone(), grid_buy_count: grid_buy_count.clone(),
                grid_rate, grid_offset: grid_offset_256,
76          first_base_amount: first_base_amount_256, first_quote_amount: first_quote_amount_256,
                last_base_amount: last_base_amount_256,
```

```
77          last_quote_amount: last_quote_amount_256, fill_base_or_quote, trigger_price:
                trigger_price_256, trigger_price_above_or_below: false,
78          take_profit_price: take_profit_price_256, stop_loss_price: stop_loss_price_256,
                valid_until_time: valid_until_time_256,
79          total_quote_amount: quote_amount_buy, total_base_amount: base_amount_sell, revenue:
                U256C::from(0), total_revenue: U256C::from(0)
80      };
81
82
83      if self.internal_need_wrap_near(&user, &pair, base_amount_sell, quote_amount_buy) {
84          // wrap near to wnear first
85          let bot_near_amount = self.internal_get_bot_near_amount(&new_grid_bot, &pair);
86          // check storage fee
87          if env::attached_deposit() - bot_near_amount < self.base_create_storage_fee + self.
                per_grid_storage_fee * (grid_buy_count + grid_sell_count) as u128 {
88              self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                    LESS_STORAGE_FEE);
89              return;
90          }
91          self.deposit_near_to_get_wnear_for_create_bot(&pair, &user, slippage, &entry_price_256,
                 &mut new_grid_bot, bot_near_amount, recommender, env::attached_deposit() -
                bot_near_amount, initial_storage_usage);
92      } else {
93          // check storage fee
94          if env::attached_deposit() < self.base_create_storage_fee + self.per_grid_storage_fee *
                 (grid_buy_count + grid_sell_count) as u128 {
95              self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                    LESS_STORAGE_FEE);
96              return;
97          }
98          // request token price
99          if pair.require_oracle {
100             self.get_price_for_create_bot(&pair, &user, slippage, &entry_price_256, &mut
                    new_grid_bot, recommender, env::attached_deposit(), initial_storage_usage);
101         } else {
102             self.internal_create_bot(None, None, &user, slippage, &entry_price_256, &pair,
                    recommender, env::attached_deposit(), initial_storage_usage, &mut new_grid_bot)
                    ;
103         }
104     }
105 }
```

Listing 2.55: grid_bot.rs

```
15      pub fn internal_create_bot(&mut self,
16                          base_price_op: Option<Price>,
17                          quote_price_op: Option<Price>,
18                          user: &AccountId,
19                          slippage: u16,
20                          entry_price: &U256C,
21                          pair: &Pair,
22                          recommender: Option<AccountId>,
23                          storage_fee: Balance,
```

```
                        initial_storage_usage: StorageUsage,
                        grid_bot: &mut GridBot) -> bool {
    if self.status != GridStatus::Running {
        self.internal_create_bot_refund_with_near(&user, &pair, storage_fee, PAUSE_OR_SHUTDOWN)
            ;
        return false;
    }
    if pair.require_oracle && !self.internal_check_oracle_price(*entry_price, base_price_op.
        clone().unwrap().clone(), quote_price_op.clone().unwrap().clone(), slippage) {
        self.internal_create_bot_refund_with_near(&user, &pair, storage_fee, INVALID_PRICE);
        return false;
    }
    // check balance
    if self.internal_get_user_balance(user, &(pair.base_token)) < grid_bot.total_base_amount {
        self.internal_create_bot_refund_with_near(&user, &pair, storage_fee, LESS_BASE_TOKEN);
        return false;
    }
    if self.internal_get_user_balance(user, &(pair.quote_token)) < grid_bot.total_quote_amount
        {
        self.internal_create_bot_refund_with_near(&user, &pair, storage_fee, LESS_QUOTE_TOKEN);
        return false;
    }


    // create bot id
    let next_bot_id = format!("GRID:{}", self.internal_get_and_use_next_bot_id().to_string());
    grid_bot.bot_id = next_bot_id;


    // initial orders space, create empty orders
    let grid_count = grid_bot.grid_sell_count.clone() + grid_bot.grid_buy_count.clone();
    self.create_default_orders(grid_bot.bot_id.clone(), grid_count);


    // transfer assets
    self.internal_transfer_assets_to_lock(&user, &pair.base_token, grid_bot.total_base_amount);
    self.internal_transfer_assets_to_lock(&user, &pair.quote_token, grid_bot.total_quote_amount
        );


    // init active status of bot
    self.internal_init_bot_status(grid_bot, entry_price);


    // insert bot
    self.bot_map.insert(&(grid_bot.bot_id), &grid_bot);


    // add recommender
    self.internal_add_referral_user(recommender, &user);


    if pair.require_oracle {
```

```
73          let base_price = base_price_op.unwrap();
74          let quote_price = quote_price_op.unwrap();
75          emit::create_bot(&grid_bot.user, grid_bot.bot_id.clone(), base_price.price.0.to_string
                (), quote_price.price.0.to_string(), base_price.expo.to_string(), quote_price.expo.
                to_string(), slippage, U128::from(entry_price.as_u128()), pair.clone(), self.
                internal_get_grid_bot_output(grid_bot));
76      } else {
77          emit::create_bot(&grid_bot.user, grid_bot.bot_id.clone(), "0".to_string(), "0".
                to_string(), "0".to_string(), "0".to_string(), slippage, U128::from(entry_price.
                as_u128()), pair.clone(), self.internal_get_grid_bot_output(grid_bot));
78      }
79
80
81      self.internal_refund_deposit(storage_fee, initial_storage_usage, &user);
82      return true;
83  }
```

Listing 2.56: grid_bot_internal.rs

```
178   pub fn internal_add_referral_user(&mut self, recommender_op: Option<AccountId>, user: &
          AccountId) {
179       if self.refer_user_recommender_map.contains_key(user) || recommender_op.is_none() || user.
              clone() == recommender_op.clone().unwrap() {
180           return;
181       }
182       let recommender = recommender_op.unwrap();
183       self.internal_add_refer(user, &recommender);
184       emit::add_referral(user, &recommender);
185   }
```

Listing 2.57: grid_bot.rs

```
295   pub fn internal_add_refer(&mut self, user: &AccountId, recommender: &AccountId) {
296       self.internal_add_refer_user_recommend(user, recommender);
297       self.internal_add_refer_recommend_user(user, recommender);
298   }
```

Listing 2.58: grid_bot_asset.rs

```
284   pub fn internal_add_refer_recommend_user(&mut self, user: &AccountId, recommender: &AccountId)
          {
285       if !self.refer_recommender_user_map.contains_key(recommender) {
286           let key = user.to_string() + ":ref_users";
287           self.refer_recommender_user_map.insert(recommender, &Vector::new(key.as_bytes().to_vec
                  ()));
288       }
289       let mut ref_users = self.refer_recommender_user_map.get(recommender).unwrap();
290       ref_users.push(user);
291
292
293       self.refer_recommender_user_map.insert(recommender, &ref_users);
294   }
```

Listing 2.59: grid_bot_asset.rs

Impact Using the user's `accountId` as the `Storage_Key` lacks uniqueness, which may result in previous data being overwritten.

Suggestion Use the `recommender`'s `accountId` as the `Storage_Key`.

2.1.21 Unrefunded Near Due to WNEAR is Not in Whitelist

Severity Medium

Status Fixed in `Version 3`

Introduced by `Version 2`

Description Any user can create a `grid_bot` through the function `create_bot()`. When `WNEAR` is needed within the `grid_bot`, this function will invoke the function `near_deposit()` contract across contracts to convert the user's attached `NEAR` into `WNEAR`. In the function `internal_deposit()`, specifically at Line 164, if `WNEAR` is not in the whitelist, it will revert directly. In this case, the user's attached `NEAR` is not refunded.

```
13      pub fn create_bot(&mut self, name: String, pair_id: String, slippage: u16, grid_type:
            GridType,
14                  grid_rate: u16, grid_offset: U128, first_base_amount: U128, first_quote_amount
                        : U128,
15                  last_base_amount: U128, last_quote_amount: U128, fill_base_or_quote: bool,
                        grid_sell_count: u16, grid_buy_count: u16,
16                  trigger_price: U128, take_profit_price: U128, stop_loss_price: U128,
                        valid_until_time: U128,
17                  entry_price: U128, recommender: Option<AccountId>) {
18      // record storage fee
19      let initial_storage_usage = env::storage_usage();
20      let grid_offset_256 = U256C::from(grid_offset.0);
21      let first_base_amount_256 = U256C::from(first_base_amount.0);
22      let first_quote_amount_256 = U256C::from(first_quote_amount.0);
23      let last_base_amount_256 = U256C::from(last_base_amount.0);
24      let last_quote_amount_256 = U256C::from(last_quote_amount.0);
25      let trigger_price_256 = U256C::from(trigger_price.0);
26      let take_profit_price_256 = U256C::from(take_profit_price.0);
27      let stop_loss_price_256 = U256C::from(stop_loss_price.0);
28      let valid_until_time_256 = U256C::from(valid_until_time.0);
29      let entry_price_256 = U256C::from(entry_price.0);
30
31
32      require!(valid_until_time.0 > env::block_timestamp_ms() as u128, INVALID_UNTIL_TIME);
33
34
35      require!(self.pair_map.contains_key(&pair_id), INVALID_PAIR_ID);
36      let pair = self.pair_map.get(&pair_id).unwrap().clone();
37      let user = env::predecessor_account_id();
38
39
40      // require!(self.status == GridStatus::Running, PAUSE_OR_SHUTDOWN);
41      if self.status != GridStatus::Running {
42          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
```

```
43          return;
44      }
45
46
47      if grid_buy_count + grid_sell_count > MAX_GRID_COUNT {
48          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                MORE_THAN_MAX_GRID_COUNT);
49          return;
50      }
51
52
53      // calculate all assets
54      let (base_amount_sell, quote_amount_buy) = GridBotContract::internal_calculate_bot_assets(
            first_quote_amount_256.clone(), last_base_amount_256.clone(), grid_sell_count.clone(),
55                  grid_buy_count.clone(),grid_type.clone(), grid_rate.clone(), grid_offset_256.
                        clone(), fill_base_or_quote.clone());
56
57
58      // require!(env::attached_deposit() >= STORAGE_FEE, LESS_STORAGE_FEE);
59      if !self.internal_check_near_amount(&user, &pair, env::attached_deposit(), base_amount_sell
            , quote_amount_buy) {
60          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                INVALID_AMOUNT);
61          return;
62      }
63      // last_quote_amount / last_base_amount > first_quote_amount > first_base_amount
64      // amount must u128, u128 * u128 <= u256, so, it's ok
65      let (result, reason) = self.internal_check_bot_amount(grid_sell_count, grid_buy_count,
            first_base_amount_256, first_quote_amount_256,
66                                          last_base_amount_256, last_quote_amount_256, &pair,
                                                base_amount_sell, quote_amount_buy);
67      if !result {
68          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(), &
                reason);
69          return;
70      }
71
72
73      // create bot
74      let mut new_grid_bot = GridBot {name, active: false, user: user.clone(), bot_id: "".
            to_string(), closed: false, pair_id, grid_type,
75          grid_sell_count: grid_sell_count.clone(), grid_buy_count: grid_buy_count.clone(),
                grid_rate, grid_offset: grid_offset_256,
76          first_base_amount: first_base_amount_256, first_quote_amount: first_quote_amount_256,
                last_base_amount: last_base_amount_256,
77          last_quote_amount: last_quote_amount_256, fill_base_or_quote, trigger_price:
                trigger_price_256, trigger_price_above_or_below: false,
78          take_profit_price: take_profit_price_256, stop_loss_price: stop_loss_price_256,
                valid_until_time: valid_until_time_256,
79          total_quote_amount: quote_amount_buy, total_base_amount: base_amount_sell, revenue:
                U256C::from(0), total_revenue: U256C::from(0)
80      };
81
```

```
82
83      if self.internal_need_wrap_near(&user, &pair, base_amount_sell, quote_amount_buy) {
84          // wrap near to wnear first
85          let bot_near_amount = self.internal_get_bot_near_amount(&new_grid_bot, &pair);
86          // check storage fee
87          if env::attached_deposit() - bot_near_amount < self.base_create_storage_fee + self.
                per_grid_storage_fee * (grid_buy_count + grid_sell_count) as u128 {
88              self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                    LESS_STORAGE_FEE);
89              return;
90          }
91          self.deposit_near_to_get_wnear_for_create_bot(&pair, &user, slippage, &entry_price_256,
                 &mut new_grid_bot, bot_near_amount, recommender, env::attached_deposit() -
                bot_near_amount, initial_storage_usage);
92      } else {
93          // check storage fee
94          if env::attached_deposit() < self.base_create_storage_fee + self.per_grid_storage_fee *
                 (grid_buy_count + grid_sell_count) as u128 {
95              self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                    LESS_STORAGE_FEE);
96              return;
97          }
98          // request token price
99          if pair.require_oracle {
100             self.get_price_for_create_bot(&pair, &user, slippage, &entry_price_256, &mut
                    new_grid_bot, recommender, env::attached_deposit(), initial_storage_usage);
101         } else {
102             self.internal_create_bot(None, None, &user, slippage, &entry_price_256, &pair,
                    recommender, env::attached_deposit(), initial_storage_usage, &mut new_grid_bot)
                    ;
103         }
104     }
105 }
```

Listing 2.60: grid_bot.rs

```
65  fn after_wrap_near_for_create_bot(&mut self, pair: &Pair, user: &AccountId, slippage: u16,
        entry_price: &U256C, grid_bot: &mut GridBot, amount: u128, recommender: Option<AccountId>,
         storage_fee: u128, storage_used: StorageUsage) -> bool {
66      let promise_success = is_promise_success();
67      if !promise_success.clone() {
68          // refund token and near
69          self.internal_create_bot_refund_with_near(user, pair, amount + storage_fee,
                WRAP_TO_WNEAR_ERROR);
70          emit::wrap_near_error(user, 0, amount, true);
71      } else {
72          // deposit
73          if !self.internal_deposit(&user.clone(), &self.wnear.clone(), U128::from(amount)) {
74              // maybe just need hande one token, but it's ok, no problem
75              self.internal_increase_asset(user, &self.wnear.clone(), &U256C::from(amount.clone()
                    ));
76              self.internal_create_bot_refund_with_near(user, pair, storage_fee,
                    WRAP_TO_WNEAR_ERROR);
```

```
77              emit::wrap_near_error(user, 0, amount, true);
78          } else {
79              // request price
80              // reduce storage fee, because deposit
81              let new_storage_fee = storage_fee - self.storage_price_per_byte * ((env::
                    storage_usage() - storage_used) as u128);
82              if pair.require_oracle {
83                  self.get_price_for_create_bot(pair, user, slippage, entry_price, grid_bot,
                        recommender, new_storage_fee, storage_used);
84              } else {
85                  self.internal_create_bot(None, None, user, slippage, entry_price, pair,
                        recommender, new_storage_fee, storage_used, grid_bot);
86              }
87          }
88      }
89      promise_success
90  }
```

Listing 2.61: wnear.rs

```
163 pub fn internal_deposit(&mut self, sender_id: &AccountId, token_in: &AccountId, amount: U128)
        -> bool {
164     require!(self.global_balances_map.contains_key(token_in), INVALID_TOKEN);
165     if !self.query_user_token_registered(sender_id.clone(), token_in.clone()) {
166         emit::deposit_failed(sender_id, amount.clone().0, token_in);
167         return false;
168     }
169     // require min deposit
170     if amount.clone().0 < self.deposit_limit_map.get(token_in).unwrap().as_u128() {
171         emit::deposit_failed(sender_id, amount.clone().0, token_in);
172         return false;
173     }
174     // log!("Deposit user:{}, token:{}, amount:{}", sender_id.clone(), token_in.clone(), amount
            .clone().0);
175     // add amount to user
176     self.internal_increase_asset(sender_id, token_in, &(U256C::from(amount.clone().0)));
177     // add amount to global
178     self.internal_increase_global_asset(token_in, &(U256C::from(amount.clone().0)));
179     // event
180     emit::deposit_success(sender_id, amount.clone().0, token_in);
181     return true;
182 }
```

Listing 2.62: grid_bot_asset.rs

Impact When `WNEAR` is not on the contract's whitelist, the `NEAR` attached by the user when creating a `grid_bot` is not refunded.

Suggestion Add logic to handle refunds or ensure that `WNEAR` is on the whitelist.

2.1.22 Incorrect Storage Fee Logic (I)

Severity High

Status Fixed in `Version 3`

Introduced by `Version 2`

Description Any user can invoke the function `create_bot()` to create a `grid_bot`. If the `grid_bot` contains `WNEAR`, the contract will invoke the function `near_deposit()` to convert `NEAR` into `WNEAR`, then record the user's `WNEAR` balance through the function `internal_deposit()`. An error occurs in the calculation of the user's storage_fee within the function `after_wrap_near_for_create_bot()`. Specifically, from the entry in `create_bot()` to line 81 in `after_wrap_near_for_create_bot()`, the user has not added any new `storage_usage`. Furthermore, due to a cross-contract call involved in the transaction, which does not execute within a single block, the value obtained from `env::storage_usage()` may be overstated as it includes the additional `storage_usage` from other users.

```
13      pub fn create_bot(&mut self, name: String, pair_id: String, slippage: u16, grid_type:
            GridType,
14                  grid_rate: u16, grid_offset: U128, first_base_amount: U128, first_quote_amount
                        : U128,
15                  last_base_amount: U128, last_quote_amount: U128, fill_base_or_quote: bool,
                        grid_sell_count: u16, grid_buy_count: u16,
16                  trigger_price: U128, take_profit_price: U128, stop_loss_price: U128,
                        valid_until_time: U128,
17                  entry_price: U128, recommender: Option<AccountId>) {
18      // record storage fee
19      let initial_storage_usage = env::storage_usage();
20      let grid_offset_256 = U256C::from(grid_offset.0);
21      let first_base_amount_256 = U256C::from(first_base_amount.0);
22      let first_quote_amount_256 = U256C::from(first_quote_amount.0);
23      let last_base_amount_256 = U256C::from(last_base_amount.0);
24      let last_quote_amount_256 = U256C::from(last_quote_amount.0);
25      let trigger_price_256 = U256C::from(trigger_price.0);
26      let take_profit_price_256 = U256C::from(take_profit_price.0);
27      let stop_loss_price_256 = U256C::from(stop_loss_price.0);
28      let valid_until_time_256 = U256C::from(valid_until_time.0);
29      let entry_price_256 = U256C::from(entry_price.0);
30
31
32      require!(valid_until_time.0 > env::block_timestamp_ms() as u128, INVALID_UNTIL_TIME);
33
34
35      require!(self.pair_map.contains_key(&pair_id), INVALID_PAIR_ID);
36      let pair = self.pair_map.get(&pair_id).unwrap().clone();
37      let user = env::predecessor_account_id();
38
39
40      // require!(self.status == GridStatus::Running, PAUSE_OR_SHUTDOWN);
41      if self.status != GridStatus::Running {
42          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
43          return;
44      }
45
46
```

```
47      if grid_buy_count + grid_sell_count > MAX_GRID_COUNT {
48          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                MORE_THAN_MAX_GRID_COUNT);
49          return;
50      }
51
52
53      // calculate all assets
54      let (base_amount_sell, quote_amount_buy) = GridBotContract::internal_calculate_bot_assets(
            first_quote_amount_256.clone(), last_base_amount_256.clone(), grid_sell_count.clone(),
55                grid_buy_count.clone(),grid_type.clone(), grid_rate.clone(), grid_offset_256.
                      clone(), fill_base_or_quote.clone());
56
57
58      // require!(env::attached_deposit() >= STORAGE_FEE, LESS_STORAGE_FEE);
59      if !self.internal_check_near_amount(&user, &pair, env::attached_deposit(), base_amount_sell
            , quote_amount_buy) {
60          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                INVALID_AMOUNT);
61          return;
62      }
63      // last_quote_amount / last_base_amount > first_quote_amount > first_base_amount
64      // amount must u128, u128 * u128 <= u256, so, it's ok
65      let (result, reason) = self.internal_check_bot_amount(grid_sell_count, grid_buy_count,
            first_base_amount_256, first_quote_amount_256,
66                                          last_base_amount_256, last_quote_amount_256, &pair,
                                                base_amount_sell, quote_amount_buy);
67      if !result {
68          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(), &
                reason);
69          return;
70      }
71
72
73      // create bot
74      let mut new_grid_bot = GridBot {name, active: false, user: user.clone(), bot_id: "".
            to_string(), closed: false, pair_id, grid_type,
75          grid_sell_count: grid_sell_count.clone(), grid_buy_count: grid_buy_count.clone(),
                grid_rate, grid_offset: grid_offset_256,
76          first_base_amount: first_base_amount_256, first_quote_amount: first_quote_amount_256,
                last_base_amount: last_base_amount_256,
77          last_quote_amount: last_quote_amount_256, fill_base_or_quote, trigger_price:
                trigger_price_256, trigger_price_above_or_below: false,
78          take_profit_price: take_profit_price_256, stop_loss_price: stop_loss_price_256,
                valid_until_time: valid_until_time_256,
79          total_quote_amount: quote_amount_buy, total_base_amount: base_amount_sell, revenue:
                U256C::from(0), total_revenue: U256C::from(0)
80      };
81
82
83      if self.internal_need_wrap_near(&user, &pair, base_amount_sell, quote_amount_buy) {
84          // wrap near to wnear first
85          let bot_near_amount = self.internal_get_bot_near_amount(&new_grid_bot, &pair);
```

```
86          // check storage fee
87          if env::attached_deposit() - bot_near_amount < self.base_create_storage_fee + self.
                per_grid_storage_fee * (grid_buy_count + grid_sell_count) as u128 {
88              self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                    LESS_STORAGE_FEE);
89              return;
90          }
91          self.deposit_near_to_get_wnear_for_create_bot(&pair, &user, slippage, &entry_price_256,
                 &mut new_grid_bot, bot_near_amount, recommender, env::attached_deposit() -
                bot_near_amount, initial_storage_usage);
92      } else {
93          // check storage fee
94          if env::attached_deposit() < self.base_create_storage_fee + self.per_grid_storage_fee *
                 (grid_buy_count + grid_sell_count) as u128 {
95              self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                    LESS_STORAGE_FEE);
96              return;
97          }
98          // request token price
99          if pair.require_oracle {
100             self.get_price_for_create_bot(&pair, &user, slippage, &entry_price_256, &mut
                    new_grid_bot, recommender, env::attached_deposit(), initial_storage_usage);
101         } else {
102             self.internal_create_bot(None, None, &user, slippage, &entry_price_256, &pair,
                    recommender, env::attached_deposit(), initial_storage_usage, &mut new_grid_bot)
                    ;
103         }
104     }
105 }
```

Listing 2.63: grid_bot.rs

```
29  pub fn deposit_near_to_get_wnear_for_create_bot(&mut self, pair: &Pair, user: &AccountId,
        slippage: u16, entry_price: &U256C,
30                              grid_bot: &mut GridBot, amount: u128, recommender: Option<
                                    AccountId>, storage_fee: u128, storage_used: StorageUsage) {
31      ext_wnear::ext(self.wnear.clone())
32          .with_attached_deposit(amount)
33          // .with_static_gas(GAS_FOR_CREATE_BOT_AFTER_NEAR)
34          .near_deposit()
35          .then(
36          Self::ext(env::current_account_id())
37              .with_static_gas(GAS_FOR_CREATE_BOT_AFTER_NEAR)
38              .after_wrap_near_for_create_bot(
39                  pair,
40                  user,
41                  slippage,
42                  entry_price,
43                  grid_bot,
44                  amount,
45                  recommender,
46                  storage_fee,
47                  storage_used
```

```
48              )
49      );
50  }
```

Listing 2.64: wnear.rs

```
65  fn after_wrap_near_for_create_bot(&mut self, pair: &Pair, user: &AccountId, slippage: u16,
        entry_price: &U256C, grid_bot: &mut GridBot, amount: u128, recommender: Option<AccountId>,
         storage_fee: u128, storage_used: StorageUsage) -> bool {
66      let promise_success = is_promise_success();
67      if !promise_success.clone() {
68          // refund token and near
69          self.internal_create_bot_refund_with_near(user, pair, amount + storage_fee,
                WRAP_TO_WNEAR_ERROR);
70          emit::wrap_near_error(user, 0, amount, true);
71      } else {
72          // deposit
73          if !self.internal_deposit(&user.clone(), &self.wnear.clone(), U128::from(amount)) {
74              // maybe just need hande one token, but it's ok, no problem
75              self.internal_increase_asset(user, &self.wnear.clone(), &U256C::from(amount.clone()
                    ));
76              self.internal_create_bot_refund_with_near(user, pair, storage_fee,
                    WRAP_TO_WNEAR_ERROR);
77              emit::wrap_near_error(user, 0, amount, true);
78          } else {
79              // request price
80              // reduce storage fee, because deposit
81              let new_storage_fee = storage_fee - self.storage_price_per_byte * ((env::
                    storage_usage() - storage_used) as u128);
82              if pair.require_oracle {
83                  self.get_price_for_create_bot(pair, user, slippage, entry_price, grid_bot,
                        recommender, new_storage_fee, storage_used);
84              } else {
85                  self.internal_create_bot(None, None, user, slippage, entry_price, pair,
                        recommender, new_storage_fee, storage_used, grid_bot);
86              }
87          }
88      }
89      promise_success
90  }
```

Listing 2.65: wnear.rs

Impact Due to incorrect storage fee calculations, users may overpay for storage fees.

Suggestion Ensure that when calculating changes in a user's `storage_usage`, the `env::storage_usage()` values obtained at the start and end do not span across blocks. Additionally, clearly identify parts of the code that will increase `storage_usage`.

2.1.23 Incorrect Storage Fee Logic (II)

Severity High

Status Fixed in `Version 3`

Introduced by Version 2

Description In function `internal_create_bot()`, the `initial_storage_usage` is obtained in function `create_bot()` while the `reserved_storage_fee` is updated in `after_wrap_near_for_create_bot()`, which is inconsistent. Consequently, the required storage fee from the commencement of `create_bot()` up to line 81 in function `after_wrap_near_for_create_bot()` is paid twice.

```
13      pub fn create_bot(&mut self, name: String, pair_id: String, slippage: u16, grid_type:
            GridType,
14                  grid_rate: u16, grid_offset: U128, first_base_amount: U128, first_quote_amount
                        : U128,
15                  last_base_amount: U128, last_quote_amount: U128, fill_base_or_quote: bool,
                        grid_sell_count: u16, grid_buy_count: u16,
16                  trigger_price: U128, take_profit_price: U128, stop_loss_price: U128,
                        valid_until_time: U128,
17                  entry_price: U128, recommender: Option<AccountId>) {
18      // record storage fee
19      let initial_storage_usage = env::storage_usage();
20      let grid_offset_256 = U256C::from(grid_offset.0);
21      let first_base_amount_256 = U256C::from(first_base_amount.0);
22      let first_quote_amount_256 = U256C::from(first_quote_amount.0);
23      let last_base_amount_256 = U256C::from(last_base_amount.0);
24      let last_quote_amount_256 = U256C::from(last_quote_amount.0);
25      let trigger_price_256 = U256C::from(trigger_price.0);
26      let take_profit_price_256 = U256C::from(take_profit_price.0);
27      let stop_loss_price_256 = U256C::from(stop_loss_price.0);
28      let valid_until_time_256 = U256C::from(valid_until_time.0);
29      let entry_price_256 = U256C::from(entry_price.0);
30
31
32      require!(valid_until_time.0 > env::block_timestamp_ms() as u128, INVALID_UNTIL_TIME);
33
34
35      require!(self.pair_map.contains_key(&pair_id), INVALID_PAIR_ID);
36      let pair = self.pair_map.get(&pair_id).unwrap().clone();
37      let user = env::predecessor_account_id();
38
39
40      // require!(self.status == GridStatus::Running, PAUSE_OR_SHUTDOWN);
41      if self.status != GridStatus::Running {
42          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
43          return;
44      }
45
46
47      if grid_buy_count + grid_sell_count > MAX_GRID_COUNT {
48          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                MORE_THAN_MAX_GRID_COUNT);
49          return;
50      }
51
52
```

```
53      // calculate all assets
54      let (base_amount_sell, quote_amount_buy) = GridBotContract::internal_calculate_bot_assets(
            first_quote_amount_256.clone(), last_base_amount_256.clone(), grid_sell_count.clone(),
55                  grid_buy_count.clone(),grid_type.clone(), grid_rate.clone(), grid_offset_256.
                        clone(), fill_base_or_quote.clone());
56
57
58      // require!(env::attached_deposit() >= STORAGE_FEE, LESS_STORAGE_FEE);
59      if !self.internal_check_near_amount(&user, &pair, env::attached_deposit(), base_amount_sell
            , quote_amount_buy) {
60          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                INVALID_AMOUNT);
61          return;
62      }
63      // last_quote_amount / last_base_amount > first_quote_amount > first_base_amount
64      // amount must u128, u128 * u128 <= u256, so, it's ok
65      let (result, reason) = self.internal_check_bot_amount(grid_sell_count, grid_buy_count,
            first_base_amount_256, first_quote_amount_256,
66                                          last_base_amount_256, last_quote_amount_256, &pair,
                                                base_amount_sell, quote_amount_buy);
67      if !result {
68          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(), &
                reason);
69          return;
70      }
71
72
73      // create bot
74      let mut new_grid_bot = GridBot {name, active: false, user: user.clone(), bot_id: "".
            to_string(), closed: false, pair_id, grid_type,
75          grid_sell_count: grid_sell_count.clone(), grid_buy_count: grid_buy_count.clone(),
                grid_rate, grid_offset: grid_offset_256,
76          first_base_amount: first_base_amount_256, first_quote_amount: first_quote_amount_256,
                last_base_amount: last_base_amount_256,
77          last_quote_amount: last_quote_amount_256, fill_base_or_quote, trigger_price:
                trigger_price_256, trigger_price_above_or_below: false,
78          take_profit_price: take_profit_price_256, stop_loss_price: stop_loss_price_256,
                valid_until_time: valid_until_time_256,
79          total_quote_amount: quote_amount_buy, total_base_amount: base_amount_sell, revenue:
                U256C::from(0), total_revenue: U256C::from(0)
80      };
81
82
83      if self.internal_need_wrap_near(&user, &pair, base_amount_sell, quote_amount_buy) {
84          // wrap near to wnear first
85          let bot_near_amount = self.internal_get_bot_near_amount(&new_grid_bot, &pair);
86          // check storage fee
87          if env::attached_deposit() - bot_near_amount < self.base_create_storage_fee + self.
                per_grid_storage_fee * (grid_buy_count + grid_sell_count) as u128 {
88              self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                    LESS_STORAGE_FEE);
89              return;
90          }
```

```
91          self.deposit_near_to_get_wnear_for_create_bot(&pair, &user, slippage, &entry_price_256,
                &mut new_grid_bot, bot_near_amount, recommender, env::attached_deposit() -
                bot_near_amount, initial_storage_usage);
92      } else {
93          // check storage fee
94          if env::attached_deposit() < self.base_create_storage_fee + self.per_grid_storage_fee *
                (grid_buy_count + grid_sell_count) as u128 {
95              self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                    LESS_STORAGE_FEE);
96              return;
97          }
98          // request token price
99          if pair.require_oracle {
100             self.get_price_for_create_bot(&pair, &user, slippage, &entry_price_256, &mut
                    new_grid_bot, recommender, env::attached_deposit(), initial_storage_usage);
101         } else {
102             self.internal_create_bot(None, None, &user, slippage, &entry_price_256, &pair,
                    recommender, env::attached_deposit(), initial_storage_usage, &mut new_grid_bot)
                    ;
103         }
104     }
105 }
```

Listing 2.66: grid_bot.rs

```
29  pub fn deposit_near_to_get_wnear_for_create_bot(&mut self, pair: &Pair, user: &AccountId,
        slippage: u16, entry_price: &U256C,
30                              grid_bot: &mut GridBot, amount: u128, recommender: Option<
                                    AccountId>, storage_fee: u128, storage_used: StorageUsage) {
31      ext_wnear::ext(self.wnear.clone())
32          .with_attached_deposit(amount)
33          // .with_static_gas(GAS_FOR_CREATE_BOT_AFTER_NEAR)
34          .near_deposit()
35          .then(
36          Self::ext(env::current_account_id())
37              .with_static_gas(GAS_FOR_CREATE_BOT_AFTER_NEAR)
38              .after_wrap_near_for_create_bot(
39                  pair,
40                  user,
41                  slippage,
42                  entry_price,
43                  grid_bot,
44                  amount,
45                  recommender,
46                  storage_fee,
47                  storage_used
48              )
49      );
50  }
```

Listing 2.67: wnear.rs

```
65  fn after_wrap_near_for_create_bot(&mut self, pair: &Pair, user: &AccountId, slippage: u16,
        entry_price: &U256C, grid_bot: &mut GridBot, amount: u128, recommender: Option<AccountId>,
         storage_fee: u128, storage_used: StorageUsage) -> bool {
66      let promise_success = is_promise_success();
67      if !promise_success.clone() {
68          // refund token and near
69          self.internal_create_bot_refund_with_near(user, pair, amount + storage_fee,
                WRAP_TO_WNEAR_ERROR);
70          emit::wrap_near_error(user, 0, amount, true);
71      } else {
72          // deposit
73          if !self.internal_deposit(&user.clone(), &self.wnear.clone(), U128::from(amount)) {
74              // maybe just need hande one token, but it's ok, no problem
75              self.internal_increase_asset(user, &self.wnear.clone(), &U256C::from(amount.clone()
                    ));
76              self.internal_create_bot_refund_with_near(user, pair, storage_fee,
                    WRAP_TO_WNEAR_ERROR);
77              emit::wrap_near_error(user, 0, amount, true);
78          } else {
79              // request price
80              // reduce storage fee, because deposit
81              let new_storage_fee = storage_fee - self.storage_price_per_byte * ((env::
                    storage_usage() - storage_used) as u128);
82              if pair.require_oracle {
83                  self.get_price_for_create_bot(pair, user, slippage, entry_price, grid_bot,
                        recommender, new_storage_fee, storage_used);
84              } else {
85                  self.internal_create_bot(None, None, user, slippage, entry_price, pair,
                        recommender, new_storage_fee, storage_used, grid_bot);
86              }
87          }
88      }
89      promise_success
90  }
```

Listing 2.68: wnear.rs

```
15      pub fn internal_create_bot(&mut self,
16                             base_price_op: Option<Price>,
17                             quote_price_op: Option<Price>,
18                             user: &AccountId,
19                             slippage: u16,
20                             entry_price: &U256C,
21                             pair: &Pair,
22                             recommender: Option<AccountId>,
23                             storage_fee: Balance,
24                             initial_storage_usage: StorageUsage,
25                             grid_bot: &mut GridBot) -> bool {
26      if self.status != GridStatus::Running {
27          self.internal_create_bot_refund_with_near(&user, &pair, storage_fee, PAUSE_OR_SHUTDOWN)
                ;
28          return false;
29      }
```

```
30      if pair.require_oracle && !self.internal_check_oracle_price(*entry_price, base_price_op.
            clone().unwrap().clone(), quote_price_op.clone().unwrap().clone(), slippage) {
31          self.internal_create_bot_refund_with_near(&user, &pair, storage_fee, INVALID_PRICE);
32          return false;
33      }
34      // check balance
35      if self.internal_get_user_balance(user, &(pair.base_token)) < grid_bot.total_base_amount {
36          self.internal_create_bot_refund_with_near(&user, &pair, storage_fee, LESS_BASE_TOKEN);
37          return false;
38      }
39      if self.internal_get_user_balance(user, &(pair.quote_token)) < grid_bot.total_quote_amount
            {
40          self.internal_create_bot_refund_with_near(&user, &pair, storage_fee, LESS_QUOTE_TOKEN);
41          return false;
42      }
43
44
45      // create bot id
46      let next_bot_id = format!("GRID:{}", self.internal_get_and_use_next_bot_id().to_string());
47      grid_bot.bot_id = next_bot_id;
48
49
50      // initial orders space, create empty orders
51      let grid_count = grid_bot.grid_sell_count.clone() + grid_bot.grid_buy_count.clone();
52      self.create_default_orders(grid_bot.bot_id.clone(), grid_count);
53
54
55      // transfer assets
56      self.internal_transfer_assets_to_lock(&user, &pair.base_token, grid_bot.total_base_amount);
57      self.internal_transfer_assets_to_lock(&user, &pair.quote_token, grid_bot.total_quote_amount
            );
58
59
60      // init active status of bot
61      self.internal_init_bot_status(grid_bot, entry_price);
62
63
64      // insert bot
65      self.bot_map.insert(&(grid_bot.bot_id), &grid_bot);
66
67
68      // add recommender
69      self.internal_add_referral_user(recommender, &user);
70
71
72      if pair.require_oracle {
73          let base_price = base_price_op.unwrap();
74          let quote_price = quote_price_op.unwrap();
75          emit::create_bot(&grid_bot.user, grid_bot.bot_id.clone(), base_price.price.0.to_string
                (), quote_price.price.0.to_string(), base_price.expo.to_string(), quote_price.expo.
                to_string(), slippage, U128::from(entry_price.as_u128()), pair.clone(), self.
                internal_get_grid_bot_output(grid_bot));
76      } else {
```

```
77          emit::create_bot(&grid_bot.user, grid_bot.bot_id.clone(), "0".to_string(), "0".
                to_string(), "0".to_string(), "0".to_string(), slippage, U128::from(entry_price.
                as_u128()), pair.clone(), self.internal_get_grid_bot_output(grid_bot));
78      }
79
80
81      self.internal_refund_deposit(storage_fee, initial_storage_usage, &user);
82      return true;
83  }
```

Listing 2.69: grid_bot_internal.rs

```
370  pub fn internal_refund_deposit(&mut self, reserved_storage_fee: u128, initial_storage_usage:
         u64, user: &AccountId) {
371      let storage_used = env::storage_usage() - initial_storage_usage;
372      //get how much it would cost to store the information
373      let required_cost = self.storage_price_per_byte * Balance::from(storage_used);
374
375
376      //make sure that the attached deposit is greater than or equal to the required cost
377      assert!(
378          required_cost <= reserved_storage_fee,
379          "Must attach {} yoctoNEAR to cover storage",
380          required_cost,
381      );
382
383
384      //get the refund amount from the attached deposit - required cost
385      let refund = reserved_storage_fee - required_cost;
386
387
388      //if the refund is greater than 1 yocto NEAR, we refund the predecessor that amount
389      if refund > 1 {
390          self.internal_ft_transfer_near(user, refund, false);
391      }
392  }
```

Listing 2.70: grid_bot_asset.rs

Impact Due to logical error, users overpaid storage fees.

Suggestion Revise the logic accordingly.

2.1.24 Incorrect Logic in Function internal_check_near_amount()

Severity Medium

Status Fixed in `Version 3`

Introduced by `Version 2`

Description If a user's created `grid_bot` requires `WNEAR`, the function `internal_check_near_amount()` checks whether the user's `WNEAR` balance and attached `NEAR` are sufficient. However, it fails to consider the storage fee required to create the `grid_bot` when assessing the attached

NEAR, which is incorrect. Moreover, the logic from line 75 to line 83 already includes the operations of the function `internal_check_near_amount()`, rendering `internal_check_near_amount()` redundant.

```
13      pub fn create_bot(&mut self, name: String, pair_id: String, slippage: u16, grid_type:
            GridType,
14                  grid_rate: u16, grid_offset: U128, first_base_amount: U128, first_quote_amount
                        : U128,
15                  last_base_amount: U128, last_quote_amount: U128, fill_base_or_quote: bool,
                        grid_sell_count: u16, grid_buy_count: u16,
16                  trigger_price: U128, take_profit_price: U128, stop_loss_price: U128,
                        valid_until_time: U128,
17                  entry_price: U128, recommender: Option<AccountId>) {
18      // record storage fee
19      let initial_storage_usage = env::storage_usage();
20      let grid_offset_256 = U256C::from(grid_offset.0);
21      let first_base_amount_256 = U256C::from(first_base_amount.0);
22      let first_quote_amount_256 = U256C::from(first_quote_amount.0);
23      let last_base_amount_256 = U256C::from(last_base_amount.0);
24      let last_quote_amount_256 = U256C::from(last_quote_amount.0);
25      let trigger_price_256 = U256C::from(trigger_price.0);
26      let take_profit_price_256 = U256C::from(take_profit_price.0);
27      let stop_loss_price_256 = U256C::from(stop_loss_price.0);
28      let valid_until_time_256 = U256C::from(valid_until_time.0);
29      let entry_price_256 = U256C::from(entry_price.0);
30
31
32      require!(valid_until_time.0 > env::block_timestamp_ms() as u128, INVALID_UNTIL_TIME);
33
34
35      require!(self.pair_map.contains_key(&pair_id), INVALID_PAIR_ID);
36      let pair = self.pair_map.get(&pair_id).unwrap().clone();
37      let user = env::predecessor_account_id();
38
39
40      // require!(self.status == GridStatus::Running, PAUSE_OR_SHUTDOWN);
41      if self.status != GridStatus::Running {
42          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
43          return;
44      }
45
46
47      if grid_buy_count + grid_sell_count > MAX_GRID_COUNT {
48          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                MORE_THAN_MAX_GRID_COUNT);
49          return;
50      }
51
52
53      // calculate all assets
54      let (base_amount_sell, quote_amount_buy) = GridBotContract::internal_calculate_bot_assets(
            first_quote_amount_256.clone(), last_base_amount_256.clone(), grid_sell_count.clone(),
```

```
55                grid_buy_count.clone(),grid_type.clone(), grid_rate.clone(), grid_offset_256.
                      clone(), fill_base_or_quote.clone());
56
57
58        // require!(env::attached_deposit() >= STORAGE_FEE, LESS_STORAGE_FEE);
59        if !self.internal_check_near_amount(&user, &pair, env::attached_deposit(), base_amount_sell
              , quote_amount_buy) {
60            self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                  INVALID_AMOUNT);
61            return;
62        }
63        // last_quote_amount / last_base_amount > first_quote_amount > first_base_amount
64        // amount must u128, u128 * u128 <= u256, so, it's ok
65        let (result, reason) = self.internal_check_bot_amount(grid_sell_count, grid_buy_count,
              first_base_amount_256, first_quote_amount_256,
66                                                  last_base_amount_256, last_quote_amount_256, &pair,
                                                        base_amount_sell, quote_amount_buy);
67        if !result {
68            self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(), &
                  reason);
69            return;
70        }
71
72
73        // create bot
74        let mut new_grid_bot = GridBot {name, active: false, user: user.clone(), bot_id: "".
              to_string(), closed: false, pair_id, grid_type,
75            grid_sell_count: grid_sell_count.clone(), grid_buy_count: grid_buy_count.clone(),
                  grid_rate, grid_offset: grid_offset_256,
76            first_base_amount: first_base_amount_256, first_quote_amount: first_quote_amount_256,
                  last_base_amount: last_base_amount_256,
77            last_quote_amount: last_quote_amount_256, fill_base_or_quote, trigger_price:
                  trigger_price_256, trigger_price_above_or_below: false,
78            take_profit_price: take_profit_price_256, stop_loss_price: stop_loss_price_256,
                  valid_until_time: valid_until_time_256,
79            total_quote_amount: quote_amount_buy, total_base_amount: base_amount_sell, revenue:
                  U256C::from(0), total_revenue: U256C::from(0)
80        };
81
82
83        if self.internal_need_wrap_near(&user, &pair, base_amount_sell, quote_amount_buy) {
84            // wrap near to wnear first
85            let bot_near_amount = self.internal_get_bot_near_amount(&new_grid_bot, &pair);
86            // check storage fee
87            if env::attached_deposit() - bot_near_amount < self.base_create_storage_fee + self.
                  per_grid_storage_fee * (grid_buy_count + grid_sell_count) as u128 {
88                self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                      LESS_STORAGE_FEE);
89                return;
90            }
91            self.deposit_near_to_get_wnear_for_create_bot(&pair, &user, slippage, &entry_price_256,
                   &mut new_grid_bot, bot_near_amount, recommender, env::attached_deposit() -
                  bot_near_amount, initial_storage_usage);
```

```
92      } else {
93          // check storage fee
94          if env::attached_deposit() < self.base_create_storage_fee + self.per_grid_storage_fee *
                (grid_buy_count + grid_sell_count) as u128 {
95              self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                    LESS_STORAGE_FEE);
96              return;
97          }
98          // request token price
99          if pair.require_oracle {
100             self.get_price_for_create_bot(&pair, &user, slippage, &entry_price_256, &mut
                    new_grid_bot, recommender, env::attached_deposit(), initial_storage_usage);
101         } else {
102             self.internal_create_bot(None, None, &user, slippage, &entry_price_256, &pair,
                    recommender, env::attached_deposit(), initial_storage_usage, &mut new_grid_bot)
                    ;
103         }
104     }
105 }
```

Listing 2.71: grid_bot.rs

```
78  pub fn internal_check_near_amount(&mut self, user: &AccountId, pair: &Pair, near_amount: u128,
        base_amount_sell: U256C, quote_amount_buy: U256C) -> bool {
79      if pair.quote_token != self.wnear && pair.base_token != self.wnear {
80          return true;
81      }
82      let wnear_balance = self.internal_get_user_balance(&user, &self.wnear);
83      if pair.base_token == self.wnear {
84          if wnear_balance.as_u128() < base_amount_sell.as_u128() && near_amount <
                base_amount_sell.as_u128() {
85              // wnear or near balance is not enough
86              return false;
87          }
88      }
89      if pair.quote_token == self.wnear {
90          if wnear_balance.as_u128() < quote_amount_buy.as_u128() && near_amount <
                quote_amount_buy.as_u128() {
91              // wnear or near balance is not enough
92              return false;
93          }
94      }
95      return true;
96  }
```

Listing 2.72: grid_bot_check.rs

Impact When comparing the user's attached `NEAR` with the `NEAR` required by the `grid_bot`, the storage fee needed to create the bot is not considered. Additionally, the function `internal_check_near_amount()` is redundant.

Suggestion Remove the function `internal_check_near_amount()`.

2.1.25 Lack of Storage Fee in Function taker_orders()

Severity Medium

Status Fixed in `Version 3`

Introduced by `Version 2`

Description Whitelisted users of the contract can actively execute orders in the `grid_bot` through the function `take_orders()`, which generates a `maker_fee` after an order is executed. If a `recommender` was specified when the `grid_bot` was created, a portion of the `maker_fee` is awarded to the `recommender`. The function `internal_allocate_refer_fee()` records this information, thus increasing the contract's `storage_usage`. However, the function `take_orders()` does not require to attach `NEAR` as a storage fee, which is incorrect.

```
100   pub fn take_orders(&mut self, take_order: RequestOrder, maker_orders: Vec<OrderKeyInfo>) {
101       assert_one_yocto();
102       require!(self.market_user_map.contains_key(&(env::predecessor_account_id())), INVALID_USER)
              ;
103       require!(take_order.amount_sell.0 >= self.deposit_limit_map.get(&take_order.token_sell).
              unwrap().as_u128(), INVALID_AMOUNT);
104       self.internal_take_orders(&(env::predecessor_account_id()), &take_order.to_order(),
              maker_orders);
105   }
```

Listing 2.73: grid_bot.rs

```
77    pub fn internal_take_orders(&mut self, user: &AccountId, take_order: &Order, maker_orders: Vec
          <OrderKeyInfo>) -> (U256C, U256C) {
78        require!(self.status == GridStatus::Running, PAUSE_OR_SHUTDOWN);
79        require!(maker_orders.len() > 0, INVALID_MAKER_ORDERS);
80        require!(take_order.amount_sell != U256C::from(0), INVALID_ORDER_AMOUNT);
81        require!(take_order.amount_buy != U256C::from(0), INVALID_ORDER_AMOUNT);
82        require!(self.internal_get_user_balance(&user, &(take_order.token_sell)) >= take_order.
              amount_sell, LESS_TOKEN_SELL);
83        let mut took_amount_sell = U256C::from(0);
84        let mut took_amount_buy = U256C::from(0);
85        let mut took_amount_buy_with_fee = U256C::from(0);
86        let mut total_took_fee = U256C::from(0);
87        // loop take order
88        for maker_order in maker_orders.iter() {
89            if take_order.amount_sell.as_u128() == took_amount_sell.as_u128() {
90                // over
91                break;
92            }
93            let (taker_sell, taker_buy, maker, maker_fee, current_revenue, maker_left_revenue,
                  maker_total_revenue) = self.internal_take_order(maker_order.bot_id.clone(),
                  maker_order.forward_or_reverse.clone(), maker_order.level.clone(), &take_order,
                  took_amount_sell.clone(), took_amount_buy_with_fee.clone());
94            // calculate taker fee
95            let (real_taker_buy, taker_fee) = self.internal_calculate_taker_fee(taker_buy);
96            took_amount_sell += taker_sell;
97            took_amount_buy_with_fee += taker_buy;
98            took_amount_buy += real_taker_buy;
```

```
99          total_took_fee += taker_fee;
100         // send event
101         emit::take_order(user, &maker, maker_order.bot_id.clone(), maker_order.
                forward_or_reverse.clone(), maker_order.level.clone(), &taker_sell, &taker_buy, &
                maker_fee, &taker_fee, &current_revenue, &maker_left_revenue, &maker_total_revenue)
                ;
102     }
103     require!(take_order.amount_sell >= took_amount_sell, INVALID_ORDER_MATCHING);
104
105
106     // transfer taker's asset
107     self.internal_reduce_asset(&user, &(take_order.token_sell), &took_amount_sell);
108     self.internal_increase_asset(&user, &(take_order.token_buy), &took_amount_buy);
109     // add protocol fee
110     self.internal_increase_protocol_fee(&(take_order.token_buy), &(total_took_fee));
111
112
113     // log!("Success take orders, sell token:{}, buy token:{}, sell amount:{}, buy amount:{}",
            take_order.token_sell, take_order.token_buy, take_order.amount_sell, take_order.
            amount_buy);
114     return (took_amount_sell, took_amount_buy);
115 }
```

Listing 2.74: grid_bot_internal.rs

```
32  pub fn internal_take_order(&mut self, bot_id: String, forward_or_reverse: bool, level: usize,
        taker_order: &Order, took_sell: U256C, took_buy: U256C) -> (U256C, U256C, AccountId, U256C
        , U256C, U256C, U256C) {
33      let bot = self.bot_map.get(&bot_id.clone()).unwrap().clone();
34      let pair = self.pair_map.get(&bot.pair_id).unwrap().clone();
35      let (maker_order, in_orderbook) = self.query_order(bot_id.clone(), forward_or_reverse,
            level);
36      // matching check
37      GridBotContract::internal_check_order_match(maker_order.clone(), taker_order.clone());
38
39
40      // calculate
41      let (taker_sell, taker_buy, current_filled, made_order) = GridBotContract::
            internal_calculate_matching(maker_order.clone(), taker_order.clone(), took_sell,
            took_buy);
42
43
44      // place into orderbook
45      if !in_orderbook {
46          self.internal_place_order(bot_id.clone(), maker_order.clone(), forward_or_reverse.clone
                (), level.clone());
47      }
48      // update filled
49      let maker_order = self.internal_update_order_filled(bot_id.clone(), forward_or_reverse.
            clone(), level.clone(), current_filled.clone());
50      emit::order_update(bot_id.clone(), forward_or_reverse.clone(), level.clone(), &maker_order)
            ;
51
```

```
52
53      // place opposite order
54      let opposite_order = GridBotContract::internal_get_opposite_order(&made_order, bot.clone(),
            forward_or_reverse.clone(), level.clone());
55      self.internal_place_order(bot_id.clone(), opposite_order.clone(), !forward_or_reverse.clone
            (), level.clone());
56
57
58      // query real_opposite_order
59      let (real_opposite_order, _) = self.query_order(bot_id.clone(), !forward_or_reverse.clone()
            , level.clone());
60      emit::order_update(bot_id.clone(), !forward_or_reverse.clone(), level.clone(), &
            real_opposite_order);
61
62
63      // calculate bot's revenue
64      let (revenue_token, revenue, maker_fee) = self.internal_calculate_bot_revenue(
            forward_or_reverse.clone(), made_order.clone(), opposite_order);
65
66
67      // add revenue
68      // let bot_mut = self.bot_map.get_mut(&bot_id.clone()).unwrap();
69      let mut bot = self.bot_map.get(&bot_id.clone()).unwrap();
70      bot.revenue += revenue;
71      bot.total_revenue += revenue;
72      // update bot asset
73      GridBotContract::internal_update_bot_asset(&mut bot, &pair, taker_order.token_buy.clone(),
            taker_buy.as_u128(), taker_sell.as_u128());
74
75
76      // bot asset transfer
77      self.internal_reduce_locked_assets(&(bot.user), &(taker_order.token_buy), &taker_buy);
78      self.internal_increase_locked_assets(&(bot.user), &(taker_order.token_sell), &taker_sell);
79
80
81      // allocate refer fee
82      let (protocol_fee, _) = self.internal_allocate_refer_fee(&maker_fee, &bot.user, &
            revenue_token);
83      // handle protocol fee
84      self.internal_add_protocol_fee_from_revenue(&mut bot, &revenue_token, maker_fee,
            protocol_fee, &pair);
85
86
87      // update bot
88      self.bot_map.insert(&bot_id, &bot);
89
90
91      // log!("Success take order, maker bot id:{}, forward_or_reserve:{}, level:{}, took sell
            :{}, took buy:{}", bot_id, forward_or_reverse, level, taker_sell, taker_buy);
92      return (taker_sell, taker_buy, bot.user.clone(), maker_fee, revenue, bot.revenue, bot.
            total_revenue);
93  }
```

Listing 2.75: orderbook_internal.rs

```
334   pub fn internal_allocate_refer_fee(&mut self, protocol_fee: &U256C, user: &AccountId, token: &
          AccountId) -> (U256C, U256C) {
335       if protocol_fee.as_u128() == 0 {
336           return (protocol_fee.clone(), U256C::from(0));
337       }
338       let mut refer_fee = protocol_fee.as_u128();
339       let mut need_pay_fee = 0;
340       let mut pay_fee_user = user.clone();
341       let mut total_payed_fee = 0 as u128;
342       for refer_fee_rate in self.refer_fee_rate.clone() {
343           let recommender_op = self.internal_get_recommender(&pay_fee_user);
344           if recommender_op.is_none() {
345               break;
346           }
347           refer_fee = refer_fee * (refer_fee_rate as u128) / PROTOCOL_FEE_DENOMINATOR;
348           if need_pay_fee > 0 {
349               // pay to pay_fee_user
350               need_pay_fee -= refer_fee;
351               total_payed_fee += need_pay_fee;
352               // pay
353               self.internal_increase_refer_fee(&pay_fee_user, token, &U128::from(need_pay_fee));
354           }
355           need_pay_fee = refer_fee;
356           pay_fee_user = recommender_op.unwrap();
357       }
358       if need_pay_fee > 0 {
359           total_payed_fee += need_pay_fee;
360           self.internal_increase_refer_fee(&pay_fee_user, token, &U128::from(need_pay_fee));
361       }
362       return (U256C::from(protocol_fee.as_u128() - total_payed_fee), U256C::from(total_payed_fee)
              );
363   }
```

Listing 2.76: grid_bot_asset.rs

```
300   pub fn internal_increase_refer_fee(&mut self, user: &AccountId, token: &AccountId, amount: &
          U128) {
301       if amount.0 == 0 {
302           return;
303       }
304       if !self.refer_fee_map.contains_key(user) {
305           self.refer_fee_map.insert(user, &LookupMap::new(StorageKey::ReferFeeSubKey(user.clone()
                  )));
306       }
307       let mut tokens_map = self.refer_fee_map.get(user).unwrap();
308       if !tokens_map.contains_key(token) {
309           tokens_map.insert(token, &amount.clone());
310       } else {
311           tokens_map.insert(token, &U128::from(tokens_map.get(token).unwrap().0 + amount.clone()
                  .0));
```

```
312         }
313         self.refer_fee_map.insert(user, &tokens_map);
314     }
```

Listing 2.77: grid_bot_asset.rs

Impact The function `take_orders()` may increase the contract's `storage_usage`, but the storage fee is not claimed.

Suggestion Revise the logic accordingly.

2.1.26 Grid_Bot Will Never Start Due to Incorrect Parameters

Severity High

Status Fixed in `Version 3`

Introduced by `Version 2`

Description When creating a `grid_bot`, users can choose a pair that does not require an oracle. In this case, if incorrect parameters for `entry_price` and `trigger_price` are fed, the `grid_bot`'s status can be 'not started'. However, due to the condition at Line 149 in the function `trigger_bot()`, this bot will never be able to start.

```
13   pub fn create_bot(&mut self, name: String, pair_id: String, slippage: u16, grid_type: GridType
         ,
14                     grid_rate: u16, grid_offset: U128, first_base_amount: U128, first_quote_amount
                           : U128,
15                     last_base_amount: U128, last_quote_amount: U128, fill_base_or_quote: bool,
                           grid_sell_count: u16, grid_buy_count: u16,
16                     trigger_price: U128, take_profit_price: U128, stop_loss_price: U128,
                           valid_until_time: U128,
17                     entry_price: U128, recommender: Option<AccountId>) {
18         // record storage fee
19         let initial_storage_usage = env::storage_usage();
20         let grid_offset_256 = U256C::from(grid_offset.0);
21         let first_base_amount_256 = U256C::from(first_base_amount.0);
22         let first_quote_amount_256 = U256C::from(first_quote_amount.0);
23         let last_base_amount_256 = U256C::from(last_base_amount.0);
24         let last_quote_amount_256 = U256C::from(last_quote_amount.0);
25         let trigger_price_256 = U256C::from(trigger_price.0);
26         let take_profit_price_256 = U256C::from(take_profit_price.0);
27         let stop_loss_price_256 = U256C::from(stop_loss_price.0);
28         let valid_until_time_256 = U256C::from(valid_until_time.0);
29         let entry_price_256 = U256C::from(entry_price.0);
30
31
32         require!(valid_until_time.0 > env::block_timestamp_ms() as u128, INVALID_UNTIL_TIME);
33
34
35         require!(self.pair_map.contains_key(&pair_id), INVALID_PAIR_ID);
36         let pair = self.pair_map.get(&pair_id).unwrap().clone();
37         let user = env::predecessor_account_id();
38
```

```
39
40      // require!(self.status == GridStatus::Running, PAUSE_OR_SHUTDOWN);
41      if self.status != GridStatus::Running {
42          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
43          return;
44      }
45
46
47      if grid_buy_count + grid_sell_count > MAX_GRID_COUNT {
48          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                MORE_THAN_MAX_GRID_COUNT);
49          return;
50      }
51
52
53      // calculate all assets
54      let (base_amount_sell, quote_amount_buy) = GridBotContract::internal_calculate_bot_assets(
            first_quote_amount_256.clone(), last_base_amount_256.clone(), grid_sell_count.clone(),
55                grid_buy_count.clone(),grid_type.clone(), grid_rate.clone(), grid_offset_256.
                    clone(), fill_base_or_quote.clone());
56
57
58      // require!(env::attached_deposit() >= STORAGE_FEE, LESS_STORAGE_FEE);
59      if !self.internal_check_near_amount(&user, &pair, env::attached_deposit(), base_amount_sell
            , quote_amount_buy) {
60          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                INVALID_AMOUNT);
61          return;
62      }
63      // last_quote_amount / last_base_amount > first_quote_amount > first_base_amount
64      // amount must u128, u128 * u128 <= u256, so, it's ok
65      let (result, reason) = self.internal_check_bot_amount(grid_sell_count, grid_buy_count,
            first_base_amount_256, first_quote_amount_256,
66                                          last_base_amount_256, last_quote_amount_256, &pair,
                                                base_amount_sell, quote_amount_buy);
67      if !result {
68          self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(), &
                reason);
69          return;
70      }
71
72
73      // create bot
74      let mut new_grid_bot = GridBot {name, active: false, user: user.clone(), bot_id: "".
            to_string(), closed: false, pair_id, grid_type,
75          grid_sell_count: grid_sell_count.clone(), grid_buy_count: grid_buy_count.clone(),
                grid_rate, grid_offset: grid_offset_256,
76          first_base_amount: first_base_amount_256, first_quote_amount: first_quote_amount_256,
                last_base_amount: last_base_amount_256,
77          last_quote_amount: last_quote_amount_256, fill_base_or_quote, trigger_price:
                trigger_price_256, trigger_price_above_or_below: false,
78          take_profit_price: take_profit_price_256, stop_loss_price: stop_loss_price_256,
```

```
                valid_until_time: valid_until_time_256,
79          total_quote_amount: quote_amount_buy, total_base_amount: base_amount_sell, revenue:
                U256C::from(0), total_revenue: U256C::from(0)
80      };
81
82
83      if self.internal_need_wrap_near(&user, &pair, base_amount_sell, quote_amount_buy) {
84          // wrap near to wnear first
85          let bot_near_amount = self.internal_get_bot_near_amount(&new_grid_bot, &pair);
86          // check storage fee
87          if env::attached_deposit() - bot_near_amount < self.base_create_storage_fee + self.
                per_grid_storage_fee * (grid_buy_count + grid_sell_count) as u128 {
88              self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                    LESS_STORAGE_FEE);
89              return;
90          }
91          self.deposit_near_to_get_wnear_for_create_bot(&pair, &user, slippage, &entry_price_256,
                 &mut new_grid_bot, bot_near_amount, recommender, env::attached_deposit() -
                bot_near_amount, initial_storage_usage);
92      } else {
93          // check storage fee
94          if env::attached_deposit() < self.base_create_storage_fee + self.per_grid_storage_fee *
                 (grid_buy_count + grid_sell_count) as u128 {
95              self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                    LESS_STORAGE_FEE);
96              return;
97          }
98          // request token price
99          if pair.require_oracle {
100             self.get_price_for_create_bot(&pair, &user, slippage, &entry_price_256, &mut
                    new_grid_bot, recommender, env::attached_deposit(), initial_storage_usage);
101         } else {
102             self.internal_create_bot(None, None, &user, slippage, &entry_price_256, &pair,
                    recommender, env::attached_deposit(), initial_storage_usage, &mut new_grid_bot)
                    ;
103         }
104     }
105 }
```

Listing 2.78: grid_bot.rs

```
15  pub fn internal_create_bot(&mut self,
16                             base_price_op: Option<Price>,
17                             quote_price_op: Option<Price>,
18                             user: &AccountId,
19                             slippage: u16,
20                             entry_price: &U256C,
21                             pair: &Pair,
22                             recommender: Option<AccountId>,
23                             storage_fee: Balance,
24                             initial_storage_usage: StorageUsage,
25                             grid_bot: &mut GridBot) -> bool {
26      if self.status != GridStatus::Running {
```

```
27          self.internal_create_bot_refund_with_near(&user, &pair, storage_fee, PAUSE_OR_SHUTDOWN)
                ;
28          return false;
29      }
30      if pair.require_oracle && !self.internal_check_oracle_price(*entry_price, base_price_op.
            clone().unwrap().clone(), quote_price_op.clone().unwrap().clone(), slippage) {
31          self.internal_create_bot_refund_with_near(&user, &pair, storage_fee, INVALID_PRICE);
32          return false;
33      }
34      // check balance
35      if self.internal_get_user_balance(user, &(pair.base_token)) < grid_bot.total_base_amount {
36          self.internal_create_bot_refund_with_near(&user, &pair, storage_fee, LESS_BASE_TOKEN);
37          return false;
38      }
39      if self.internal_get_user_balance(user, &(pair.quote_token)) < grid_bot.total_quote_amount
            {
40          self.internal_create_bot_refund_with_near(&user, &pair, storage_fee, LESS_QUOTE_TOKEN);
41          return false;
42      }
43
44
45      // create bot id
46      let next_bot_id = format!("GRID:{}", self.internal_get_and_use_next_bot_id().to_string());
47      grid_bot.bot_id = next_bot_id;
48
49
50      // initial orders space, create empty orders
51      let grid_count = grid_bot.grid_sell_count.clone() + grid_bot.grid_buy_count.clone();
52      self.create_default_orders(grid_bot.bot_id.clone(), grid_count);
53
54
55      // transfer assets
56      self.internal_transfer_assets_to_lock(&user, &pair.base_token, grid_bot.total_base_amount);
57      self.internal_transfer_assets_to_lock(&user, &pair.quote_token, grid_bot.total_quote_amount
            );
58
59
60      // init active status of bot
61      self.internal_init_bot_status(grid_bot, entry_price);
62
63
64      // insert bot
65      self.bot_map.insert(&(grid_bot.bot_id), &grid_bot);
66
67
68      // add recommender
69      self.internal_add_referral_user(recommender, &user);
70
71
72      if pair.require_oracle {
73          let base_price = base_price_op.unwrap();
74          let quote_price = quote_price_op.unwrap();
75          emit::create_bot(&grid_bot.user, grid_bot.bot_id.clone(), base_price.price.0.to_string
```

```
                (), quote_price.price.0.to_string(), base_price.expo.to_string(), quote_price.expo.
                to_string(), slippage, U128::from(entry_price.as_u128()), pair.clone(), self.
                internal_get_grid_bot_output(grid_bot));
76      } else {
77          emit::create_bot(&grid_bot.user, grid_bot.bot_id.clone(), "0".to_string(), "0".
                to_string(), "0".to_string(), "0".to_string(), slippage, U128::from(entry_price.
                as_u128()), pair.clone(), self.internal_get_grid_bot_output(grid_bot));
78      }
79
80
81      self.internal_refund_deposit(storage_fee, initial_storage_usage, &user);
82      return true;
83  }
```

Listing 2.79: grid_bot_internal.rs

```
197  pub fn internal_init_bot_status(&self, bot: &mut GridBot, entry_price: &U256C) {
198      if bot.trigger_price == U256C::from(0) {
199          bot.active = true;
200          return;
201      }
202      if entry_price.clone() >= bot.trigger_price {
203          bot.trigger_price_above_or_below = false;
204      } else {
205          bot.trigger_price_above_or_below = true;
206      }
207  }
```

Listing 2.80: grid_bot_internal.rs

```
144  pub fn trigger_bot(&mut self, bot_id: String) {
145      require!(self.status == GridStatus::Running, PAUSE_OR_SHUTDOWN);
146      let mut bot = self.bot_map.get(&bot_id).unwrap().clone();
147      require!(bot.active.clone() == false, BOT_IS_ACTIVE);
148      let pair = self.pair_map.get(&bot.pair_id).unwrap().clone();
149      require!(pair.require_oracle, INVALID_PAIR);
150      self.get_price_for_trigger_bot(&pair, &mut bot);
151  }
```

Listing 2.81: grid_bot.rs

Impact The user's `grid_bot` cannot be triggered.

Suggestion Check the parameters properly.

2.2 Additional Recommendation

2.2.1 Redundant Code

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Function `internal_need_wrap_near()` determines whether the user's `NEAR` needs to be converted into `WNEAR`. However, the return statement at line 351 is redundant and will never be executed. The similar issue also occurs in functions `internal_reduce_asset()` and `internal_reduce_refer_fee()`.

```
333  pub fn internal_need_wrap_near(&self, user: &AccountId, pair: &Pair, base_amount: U256C,
         quote_amount: U256C) -> bool {
334      if pair.base_token != self.wnear && pair.quote_token != self.wnear {
335          return false;
336      }
337      let wnear_balance = self.internal_get_user_balance(&user, &self.wnear);
338      if pair.base_token == self.wnear {
339          // query balance
340          if wnear_balance >= base_amount {
341              return false;
342          }
343          return true
344      } else if pair.quote_token == self.wnear {
345          // query balance
346          if wnear_balance >= quote_amount {
347              return false;
348          }
349          return true
350      }
351      return true;
352  }
```

Listing 2.82: grid_bot_internal.rs

```
11  pub fn internal_reduce_asset(&mut self, user: &AccountId, token: &AccountId, amount: &U256C) {
12      let mut user_balances = self.user_balances_map.get(user).unwrap_or_else(|| {
13          let mut map = LookupMap::new(StorageKey::UserBalanceSubKey(user.clone()));
14          map.insert(token, &U256C::from(0));
15          map
16      });
17
18
19      let balance = user_balances.get(token).unwrap_or(U256C::from(0));
20      user_balances.insert(token, &(balance - amount));
21
22
23      self.user_balances_map.insert(user, &user_balances);
24  }
```

Listing 2.83: grid_bot_asset.rs

```
355  pub fn internal_reduce_refer_fee(&mut self, user: &AccountId, token: &AccountId, amount: &U128
         ) {
356      if amount.0 == 0 {
357          return;
358      }
359      if !self.refer_fee_map.contains_key(user) {
360          self.refer_fee_map.insert(user, &LookupMap::new(StorageKey::ReferFeeSubKey(user.clone()
                 )));
```

```
361         }
362         let mut tokens_map = self.refer_fee_map.get(user).unwrap();
363         require!(tokens_map.contains_key(token), INVALID_TOKEN);
364         tokens_map.insert(token, &U128::from(tokens_map.get(token).unwrap().0 - amount.clone().0));
365         self.refer_fee_map.insert(user, &tokens_map);
366     }
```

Listing 2.84: grid_bot_asset.rs

Suggestion Remove the above mentioned redundant logic.

2.2.2 Redundant Implementation of NEAR Transfer

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Callback function `after_ft_transfer_near()` is used to handle the returned promise result of transferring `NEAR`. The current implementation is redundant as even if the transfer fails and is internally recorded, the `owner` still needs to invoke privileged functions to refund. It should be more efficient to directly monitor and handle the failed transfers off-chain rather than relying on privileged functions.

```
114     pub fn internal_ft_transfer_near(&mut self, receiver_id: &AccountId, amount: Balance,
            effect_global_balance: bool) -> Promise {
115         Promise::new(receiver_id.clone()).transfer(amount)
116             .then(
117             Self::ext(env::current_account_id())
118                 .with_static_gas(GAS_FOR_AFTER_FT_TRANSFER)
119                 .after_ft_transfer_near(
120                     receiver_id.clone(),
121                     self.wnear.clone(),
122                     amount.into(),
123                     effect_global_balance,
124                 )
125             )
126     }
```

Listing 2.85: token.rs

```
209     fn after_ft_transfer_near(
210         &mut self,
211         account_id: AccountId,
212         token_id: AccountId,
213         amount: U128,
214         effect_global_balance: bool,
215     ) -> bool {
216         let promise_success = is_promise_success();
217         if !promise_success.clone() {
218             emit::withdraw_failed(&account_id, amount.clone().0, &token_id);
219             if effect_global_balance {
220                 self.internal_increase_withdraw_near_error_effect_global(&account_id, &amount);
221             } else {
```

```
222             self.internal_increase_withdraw_near_error(&account_id, &amount);
223         }
224     } else {
225         emit::withdraw_succeeded(&account_id, amount.clone().0, &token_id);
226         if effect_global_balance {
227             // reduce from global asset
228             self.internal_reduce_global_asset(&token_id, &(U256C::from(amount.clone().0)))
229         }
230     }
231     promise_success
232 }
```

Listing 2.86: token.rs

Suggestion Please refer to the following code implementation: https://github.com/linear-protocol/LiNEAR/blob/main/contracts/linear/src/internal.rs#L74

2.2.3 Lack of Minimum Value Check for taker_order.amount_sell

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Function `take_orders()` does not check the minimum value of `taker_order.amount_sell`, leading to the generation of dust orders within the protocol.

```
86  pub fn take_orders(&mut self, take_order: RequestOrder, maker_orders: Vec<OrderKeyInfo>) {
87      assert_one_yocto();
88      require!(self.market_user_map.contains_key(&(env::predecessor_account_id())), INVALID_USER)
            ;
89      self.internal_take_orders(&(env::predecessor_account_id()), &take_order.to_order(),
            maker_orders);
90  }
```

Listing 2.87: grid_bot.rs

Suggestion Add checks to ensure that the remaining tokens in the order as well as the opposite order is greater than or equal to the `deposit_limit` of the corresponding tokens.

2.2.4 Lack of Check Parameter in Function set_refer_fee_rate()

Status Fixed in `Version 3`

Introduced by `Version 2`

Description The owner can set the `refer_fee_rate` through the function `set_refer_fee_rate()`, but the function does not validate the size of the input parameter. The parameter `new_refer_fee_rate` should be less than `PROTOCOL_FEE_DENOMINATOR`.

```
316 pub fn set_refer_fee_rate(&mut self, new_refer_fee_rate: Vec<u32>) {
317     self.assert_owner();
318     self.refer_fee_rate = new_refer_fee_rate;
319 }
```

Listing 2.88: grid_bot.rs

```
334   pub fn internal_allocate_refer_fee(&mut self, protocol_fee: &U256C, user: &AccountId, token: &
          AccountId) -> (U256C, U256C) {
335       if protocol_fee.as_u128() == 0 {
336           return (protocol_fee.clone(), U256C::from(0));
337       }
338       let mut refer_fee = protocol_fee.as_u128();
339       let mut need_pay_fee = 0;
340       let mut pay_fee_user = user.clone();
341       let mut total_payed_fee = 0 as u128;
342       for refer_fee_rate in self.refer_fee_rate.clone() {
343           let recommender_op = self.internal_get_recommender(&pay_fee_user);
344           if recommender_op.is_none() {
345               break;
346           }
347           refer_fee = refer_fee * (refer_fee_rate as u128) / PROTOCOL_FEE_DENOMINATOR;
348           if need_pay_fee > 0 {
349               // pay to pay_fee_user
350               need_pay_fee -= refer_fee;
351               total_payed_fee += need_pay_fee;
352               // pay
353               self.internal_increase_refer_fee(&pay_fee_user, token, &U128::from(need_pay_fee));
354           }
355           need_pay_fee = refer_fee;
356           pay_fee_user = recommender_op.unwrap();
357       }
358       if need_pay_fee > 0 {
359           total_payed_fee += need_pay_fee;
360           self.internal_increase_refer_fee(&pay_fee_user, token, &U128::from(need_pay_fee));
361       }
362       return (U256C::from(protocol_fee.as_u128() - total_payed_fee), U256C::from(total_payed_fee)
              );
363   }
```

Listing 2.89: grid_bot_asset.rs

```
23   pub const PROTOCOL_FEE_DENOMINATOR: u128 = 1000000;
```

Listing 2.90: constants.rs

Suggestion Add check to ensure the input parameter is less than `PROTOCOL_FEE_DENOMINATOR`.

2.3 Note

2.3.1 Centralization Risks

Introduced by `Version 1`

Description In the contract `DeltaBot`, privileged account `owner` plays a critical role in governing and regulating the system-wide operations as shown below (e.g., setting various parameters, adjusting the external oracle, and registering whitelist tokens as pairs).

```
270   pub fn set_oracle(&mut self, new_oracle: AccountId) {
271       self.assert_owner();
272       self.oracle = new_oracle;
273   }
```

Listing 2.91: grid_bot.rs

```
294   pub fn set_refer_fee_rate(&mut self, new_refer_fee_rate: Vec<u32>) {
295       self.assert_owner();
296       self.refer_fee_rate = new_refer_fee_rate;
297   }
```

Listing 2.92: grid_bot.rs

```
239   pub fn register_pair(&mut self, base_token: AccountId, quote_token: AccountId,
          base_min_deposit: U128, quote_min_deposit: U128, base_oracle_id: String, quote_oracle_id:
          String) {
240       require!(env::attached_deposit() == DEFAULT_TOKEN_STORAGE_FEE * 2, LESS_TOKEN_STORAGE_FEE);
241       require!(env::predecessor_account_id() == self.owner_id, ERR_NOT_ALLOWED);
242       require!(base_token != quote_token, INVALID_TOKEN);
243       let pair_key = GridBotContract::internal_get_pair_key(base_token.clone(), quote_token.clone
              ());
244       require!(!self.pair_map.contains_key(&pair_key), PAIR_EXIST);
245       let pair = Pair{
246           base_token: base_token.clone(),
247           quote_token: quote_token.clone(),
248           base_oracle_id: self.internal_format_price_identifier(base_oracle_id),
249           quote_oracle_id: self.internal_format_price_identifier(quote_oracle_id),
250       };
251       self.pair_map.insert(&pair_key, &pair);
252       self.internal_init_token(base_token, base_min_deposit);
253       self.internal_init_token(quote_token, quote_min_deposit);
254   }
```

Listing 2.93: grid_bot.rs

2.3.2 Delayed Activation of grid_bot Due to Volatile Price Fluctuations

Introduced by Version 1

Description The project will periodically invoke the function `trigger_bot()` to trigger the `grid_bot` that meets the activation criteria. However, when the market price experiences severe fluctuations, it is possible that the price touches the `trigger_price`, but the `grid_bot` is not activated.

2.3.3 Storage Usage for Token Never Released

Introduced by Version 2

Description The contract records the user's token balance and requires the user to deposit a storage fee. However, the current code does not implement a corresponding interface for users to withdraw this portion of the storage fee when exiting the protocol. Specifically, once

a user deposits the storage fee for a token, this portion of storage will never be released, and the user cannot withdraw the corresponding storage fee.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS