Security Audit Report for DeltaBotDev

Date: Feb 23, 2024

Version: 1.0

Contact: contact@blocksec.com

BlockSec
Audit Team

Digitally signed by
BlockSec Audit Team
Date: 2024.02.26 12:33:20
+08'00'

Contents

Report Manifest

Item	Description
Client	Ref-Finance
Target	DeltaBotDev

Version History

Version	Date	Description
1.0	February 23, 2024	First Version

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 DeltaBotDev[1] of Ref-Finance.

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

Project		Commit SHA
DeltaBotDev	`Version 1`	`8962241f48e1b4852c209fb71109760c8a982874`
	`Version 2`	`730cf88fec1100ad1cbba92a8e4389a3d6a1b216`

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.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.

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

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

We show the main concrete checkpoints in the following.

1.3.1 Software Security

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

1.3.2 DeFi Security

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

1.3.3 NFT Security

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

1.3.4 Additional Recommendation

∗ Gas optimization
∗ Code quality and style

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

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. 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			
	High	High	Medium
	Low	Medium	Low
		High	*Low*
		Likelihood	

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

- High Risk: 5
- Medium Risk: 6
- Low Risk: 6
- Recommendations: 3
- Note: 2

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
19	-	Redundant Code	Recommendation	Fixed
20	-	Redundant Implementation of NEAR Transfer	Recommendation	Fixed
21	-	Lack of Minimum Value Check for taker_order.amount_sell	Recommendation	Fixed
22	-	Centralization Risks	Note	
23	-	Delayed Activation of grid_bot Due to Volatile Price Fluctuations	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
```

```
    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(),
                                                grid_type.clone(), grid_rate.clone(),
                                                    grid_offset_256.clone(), fill_base_or_quote.
                                                    clone());

    // require!(env::attached_deposit() >= STORAGE_FEE, LESS_STORAGE_FEE);
    if !self.internal_check_near_amount(&user, &pair, env::attached_deposit(), base_amount_sell
        , quote_amount_buy) {
        self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
            INVALID_AMOUNT);
        return;
    }
    // last_quote_amount / last_base_amount > first_quote_amount > first_base_amount
    // amount must u128, u128 * u128 <= u256, so, it's ok
    let (result, reason) = self.internal_check_bot_amount(grid_sell_count, grid_buy_count,
        first_base_amount_256, first_quote_amount_256,
                                                    last_base_amount_256, last_quote_amount_256,
                                                         &user, &pair, base_amount_sell,
                                                        quote_amount_buy);
    if !result {
        self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(), &
            reason);
        return;
    }

    // create bot
    let mut new_grid_bot = GridBot {name, active: false, user: user.clone(), bot_id: "".
        to_string(), closed: false, pair_id, grid_type,
        grid_sell_count: grid_sell_count.clone(), grid_buy_count: grid_buy_count.clone(),
            grid_rate, grid_offset: grid_offset_256,
        first_base_amount: first_base_amount_256, first_quote_amount: first_quote_amount_256,
            last_base_amount: last_base_amount_256,
        last_quote_amount: last_quote_amount_256, fill_base_or_quote, trigger_price:
            trigger_price_256, trigger_price_above_or_below: false,
        take_profit_price: take_profit_price_256, stop_loss_price: stop_loss_price_256,
            valid_until_time: valid_until_time_256,
        total_quote_amount: quote_amount_buy, total_base_amount: base_amount_sell, revenue:
            U256C::from(0), total_revenue: U256C::from(0)
    };

    if self.internal_need_wrap_near(&user, &pair, base_amount_sell, quote_amount_buy) {
        // wrap near to wnear first
        self.deposit_near_to_get_wnear_for_create_bot(&pair, &user, slippage, &entry_price_256,
             &mut new_grid_bot, env::attached_deposit() - STORAGE_FEE);
    } else {
        // request token price
        self.get_price_for_create_bot(&pair, &user, slippage, &entry_price_256, &mut
            new_grid_bot);
    }
}
```

Listing 2.1: grid_bot.rs

```rust

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.

```rust
pub fn withdraw_unowned_asset(&mut self, token: AccountId, to_user: AccountId) {
    self.assert_owner();
    Promise::new(token.clone())
        .function_call(
            "ft_balance_of".to_string(),
            json!({"account_id": env::current_account_id()}).to_string().into_bytes(),
            0,
            Gas(0),
        )
        .then(
            Self::ext(self.owner_id.clone())
                .with_static_gas(GAS_FOR_AFTER_FT_TRANSFER)
                .after_ft_balance_of_for_withdraw_unowned_asset(
                    token.clone(),
                    to_user,
                )
        );
}
```

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()`.

```rust
    pub fn internal_reduce_asset(&mut self, user: &AccountId, token: &AccountId, amount: &U256C) {
        let mut user_balances = self.user_balances_map.get(user).unwrap_or_else(|| {
            let mut map = LookupMap::new(StorageKey::UserBalanceSubKey(user.clone()));
            map.insert(token, &U256C::from(0));
            map
        });


        let balance = user_balances.get(token).unwrap_or(U256C::from(0));
        user_balances.insert(token, &(balance - amount));


        self.user_balances_map.insert(user, &user_balances);
    }
```

Listing 2.4: grid_bot_asset.rs

```rust
    pub fn internal_reduce_refer_fee(&mut self, user: &AccountId, token: &AccountId, amount: &U128
        ) {
        if amount.0 == 0 {
            return;
        }
        if !self.refer_fee_map.contains_key(user) {
            self.refer_fee_map.insert(user, &LookupMap::new(StorageKey::ReferFeeSubKey(user.clone()
                )));
        }
        let mut tokens_map = self.refer_fee_map.get(user).unwrap();
        require!(tokens_map.contains_key(token), INVALID_TOKEN);
        tokens_map.insert(token, &U128::from(tokens_map.get(token).unwrap().0 - amount.clone().0));
        self.refer_fee_map.insert(user, &tokens_map);
    }
```

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()`.

```rust
pub fn claim(&mut self, bot_id: String) {
    require!(self.bot_map.contains_key(&bot_id), BOT_NOT_EXIST);
    let mut bot = self.bot_map.get(&bot_id).unwrap().clone();
    let pair = self.pair_map.get(&(bot.pair_id)).unwrap().clone();
    // harvest revenue
    let (revenue_token, revenue) = self.internal_harvest_revenue(&mut bot, &pair);
    self.internal_withdraw(&(bot.user), &revenue_token, revenue);
    self.bot_map.insert(&bot_id, &bot);
    // event
    emit::claim(&env::predecessor_account_id(), &(bot.user), bot_id, &revenue_token, revenue);
}
```

Listing 2.6: grid_bot.rs

```rust
pub fn internal_withdraw(&mut self, user: &AccountId, token: &AccountId, amount: U256C) {
    if amount.as_u128() == 0 {
        return;
    }
    // reduce user asset
    self.internal_reduce_asset(user, token, &amount);
    if token.clone() == self.wnear {
        // wrap to near
        self.withdraw_near(user, amount.as_u128());
    } else {
        // start transfer
        self.internal_ft_transfer(user, token, amount.as_u128());
    }
    emit::withdraw_started(user, amount.as_u128(), token);
}
```

Listing 2.7: grid_bot_asset.rs

```rust
pub fn withdraw_near(&mut self, user: &AccountId, amount: u128) {
    ext_wnear::ext(self.wnear.clone())
        .with_attached_deposit(1)
        .near_withdraw(U128::from(amount))
        .then(
            Self::ext(env::current_account_id())
                .after_withdraw_near(
                    user,
                    amount,
                )
        );
}
```

Listing 2.8: wnear.rs

```rust
pub fn internal_ft_transfer(&mut self, account_id: &AccountId, token_id: &AccountId, amount:
     Balance) -> Promise {
    ext_fungible_token::ext(token_id.clone())
        .with_attached_deposit(ONE_YOCTO)
```

```
            .with_static_gas(GAS_FOR_FT_TRANSFER)
            .ft_transfer(
                account_id.clone(),
                amount.into(),
                None,
            ).then(
            Self::ext(env::current_account_id())
                .with_static_gas(GAS_FOR_AFTER_FT_TRANSFER)
                .after_ft_transfer(
                    account_id.clone(),
                    token_id.clone(),
                    amount.into(),
                )
        )
    }
```

Listing 2.9: token.rs

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

Listing 2.10: grid_bot_asset.rs

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

Listing 2.11: grid_bot_asset.rs

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

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`.

```rust
pub fn create_bot(&mut self, name: String, pair_id: String, slippage: u16, grid_type: GridType
    ,
                  grid_rate: u16, grid_offset: U128, first_base_amount: U128, first_quote_amount
                      : U128,
                  last_base_amount: U128, last_quote_amount: U128, fill_base_or_quote: bool,
                      grid_sell_count: u16, grid_buy_count: u16,
                  trigger_price: U128, take_profit_price: U128, stop_loss_price: U128,
                      valid_until_time: U128,
                  entry_price: U128) {
    let grid_offset_256 = U256C::from(grid_offset.0);
    let first_base_amount_256 = U256C::from(first_base_amount.0);
    let first_quote_amount_256 = U256C::from(first_quote_amount.0);
    let last_base_amount_256 = U256C::from(last_base_amount.0);
    let last_quote_amount_256 = U256C::from(last_quote_amount.0);
    let trigger_price_256 = U256C::from(trigger_price.0);
    let take_profit_price_256 = U256C::from(take_profit_price.0);
    let stop_loss_price_256 = U256C::from(stop_loss_price.0);
    let valid_until_time_256 = U256C::from(valid_until_time.0);
    let entry_price_256 = U256C::from(entry_price.0);

    require!(valid_until_time.0 > env::block_timestamp_ms() as u128, INVALID_UNTIL_TIME);

    require!(self.pair_map.contains_key(&pair_id), INVALID_PAIR_ID);
    let pair = self.pair_map.get(&pair_id).unwrap().clone();
    let user = env::predecessor_account_id();

    // require!(self.status == GridStatus::Running, PAUSE_OR_SHUTDOWN);
    if self.status != GridStatus::Running {
        self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
            PAUSE_OR_SHUTDOWN);
        return;
    }

    if grid_buy_count + grid_sell_count > MAX_GRID_COUNT {
        self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
            MORE_THAN_MAX_GRID_COUNT);
        return;
    }

    // calculate all assets
    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(),
                                                       grid_type.clone(), grid_rate.clone(),
                                                           grid_offset_256.clone(), fill_base_or_quote.
```

```
                                                    clone());

        // require!(env::attached_deposit() >= STORAGE_FEE, LESS_STORAGE_FEE);
        if !self.internal_check_near_amount(&user, &pair, env::attached_deposit(), base_amount_sell
            , quote_amount_buy) {
            self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                INVALID_AMOUNT);
            return;
        }
        // last_quote_amount / last_base_amount > first_quote_amount > first_base_amount
        // amount must u128, u128 * u128 <= u256, so, it's ok
        let (result, reason) = self.internal_check_bot_amount(grid_sell_count, grid_buy_count,
            first_base_amount_256, first_quote_amount_256,
                                                    last_base_amount_256, last_quote_amount_256,
                                                        &user, &pair, base_amount_sell,
                                                        quote_amount_buy);
        if !result {
            self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(), &
                reason);
            return;
        }

        // create bot
        let mut new_grid_bot = GridBot {name, active: false, user: user.clone(), bot_id: "".
            to_string(), closed: false, pair_id, grid_type,
            grid_sell_count: grid_sell_count.clone(), grid_buy_count: grid_buy_count.clone(),
                grid_rate, grid_offset: grid_offset_256,
            first_base_amount: first_base_amount_256, first_quote_amount: first_quote_amount_256,
                last_base_amount: last_base_amount_256,
            last_quote_amount: last_quote_amount_256, fill_base_or_quote, trigger_price:
                trigger_price_256, trigger_price_above_or_below: false,
            take_profit_price: take_profit_price_256, stop_loss_price: stop_loss_price_256,
                valid_until_time: valid_until_time_256,
            total_quote_amount: quote_amount_buy, total_base_amount: base_amount_sell, revenue:
                U256C::from(0), total_revenue: U256C::from(0)
        };

        if self.internal_need_wrap_near(&user, &pair, base_amount_sell, quote_amount_buy) {
            // wrap near to wnear first
            self.deposit_near_to_get_wnear_for_create_bot(&pair, &user, slippage, &entry_price_256,
                &mut new_grid_bot, env::attached_deposit() - STORAGE_FEE);
        } else {
            // request token price
            self.get_price_for_create_bot(&pair, &user, slippage, &entry_price_256, &mut
                new_grid_bot);
        }
    }
```

Listing 2.13: grid_bot.rs

```
    pub fn query_order(&self, bot_id: String, forward_or_reverse: bool, level: usize) -> (Order,
        bool) {
        require!(self.order_map.contains_key(&bot_id), INVALID_BOT_ID);
```

```rust
        require!(self.bot_map.contains_key(&bot_id), INVALID_BOT_ID);
        let bot = self.bot_map.get(&bot_id).unwrap();
        require!(!(bot.closed.clone()), bot.bot_id.clone() + BOT_CLOSED);
        require!(bot.active.clone(), bot.bot_id.clone() + BOT_DISABLE);
        require!(self.pair_map.contains_key(&(bot.pair_id.clone())), INVALID_PAIR_ID);
        // check timestamp
        require!(bot.valid_until_time >= U256C::from(env::block_timestamp_ms()), BOT_EXPIRED);
        let bot_orders = self.order_map.get(&bot_id).unwrap();
        let orders = if forward_or_reverse {
            bot_orders.forward_orders
        } else {
            bot_orders.reverse_orders
        };
        // check order
        require!(orders.get(level.clone() as u64).is_some(), INVALID_PARAM);
        let order = &orders.get(level as u64).unwrap();
        if GridBotContract::internal_order_is_empty(order) {
            require!(forward_or_reverse, INVALID_FORWARD_OR_REVERSE);
            // The current grid order has not been placed yet
            let pair = self.pair_map.get(&(bot.pair_id.clone())).unwrap();
            return ((GridBotContract::internal_get_first_forward_order(bot.clone(), pair.clone(),
                level.clone())), false);
        }
        return (order.clone(), true);
    }
```

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.

```rust
    pub fn create_bot(&mut self, name: String, pair_id: String, slippage: u16, grid_type: GridType
        ,
                      grid_rate: u16, grid_offset: U128, first_base_amount: U128, first_quote_amount
                          : U128,
                      last_base_amount: U128, last_quote_amount: U128, fill_base_or_quote: bool,
                          grid_sell_count: u16, grid_buy_count: u16,
                      trigger_price: U128, take_profit_price: U128, stop_loss_price: U128,
                          valid_until_time: U128,
                      entry_price: U128) {
        let grid_offset_256 = U256C::from(grid_offset.0);
        let first_base_amount_256 = U256C::from(first_base_amount.0);
        let first_quote_amount_256 = U256C::from(first_quote_amount.0);
        let last_base_amount_256 = U256C::from(last_base_amount.0);
        let last_quote_amount_256 = U256C::from(last_quote_amount.0);
        let trigger_price_256 = U256C::from(trigger_price.0);
        let take_profit_price_256 = U256C::from(take_profit_price.0);
        let stop_loss_price_256 = U256C::from(stop_loss_price.0);
        let valid_until_time_256 = U256C::from(valid_until_time.0);
        let entry_price_256 = U256C::from(entry_price.0);

        require!(valid_until_time.0 > env::block_timestamp_ms() as u128, INVALID_UNTIL_TIME);

        require!(self.pair_map.contains_key(&pair_id), INVALID_PAIR_ID);
        let pair = self.pair_map.get(&pair_id).unwrap().clone();
        let user = env::predecessor_account_id();

        // require!(self.status == GridStatus::Running, PAUSE_OR_SHUTDOWN);
        if self.status != GridStatus::Running {
            self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
            return;
        }

        if grid_buy_count + grid_sell_count > MAX_GRID_COUNT {
            self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                MORE_THAN_MAX_GRID_COUNT);
            return;
        }

        // calculate all assets
        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(),
                                                  grid_type.clone(), grid_rate.clone(),
                                                      grid_offset_256.clone(), fill_base_or_quote.
                                                      clone());

        // require!(env::attached_deposit() >= STORAGE_FEE, LESS_STORAGE_FEE);
        if !self.internal_check_near_amount(&user, &pair, env::attached_deposit(), base_amount_sell
            , quote_amount_buy) {
            self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                INVALID_AMOUNT);
            return;
        }
        // last_quote_amount / last_base_amount > first_quote_amount > first_base_amount
        // amount must u128, u128 * u128 <= u256, so, it's ok
        let (result, reason) = self.internal_check_bot_amount(grid_sell_count, grid_buy_count,
            first_base_amount_256, first_quote_amount_256,
                                                      last_base_amount_256, last_quote_amount_256,
                                                           &user, &pair, base_amount_sell,
                                                          quote_amount_buy);
        if !result {
            self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(), &
reason);
            return;
        }

        // create bot
        let mut new_grid_bot = GridBot {name, active: false, user: user.clone(), bot_id: "".
            to_string(), closed: false, pair_id, grid_type,
            grid_sell_count: grid_sell_count.clone(), grid_buy_count: grid_buy_count.clone(),
                grid_rate, grid_offset: grid_offset_256,
            first_base_amount: first_base_amount_256, first_quote_amount: first_quote_amount_256,
                last_base_amount: last_base_amount_256,
            last_quote_amount: last_quote_amount_256, fill_base_or_quote, trigger_price:
                trigger_price_256, trigger_price_above_or_below: false,
            take_profit_price: take_profit_price_256, stop_loss_price: stop_loss_price_256,
                valid_until_time: valid_until_time_256,
            total_quote_amount: quote_amount_buy, total_base_amount: base_amount_sell, revenue:
                U256C::from(0), total_revenue: U256C::from(0)
        };

        if self.internal_need_wrap_near(&user, &pair, base_amount_sell, quote_amount_buy) {
            // wrap near to wnear first
            self.deposit_near_to_get_wnear_for_create_bot(&pair, &user, slippage, &entry_price_256,
                 &mut new_grid_bot, env::attached_deposit() - STORAGE_FEE);
        } else {
            // request token price
            self.get_price_for_create_bot(&pair, &user, slippage, &entry_price_256, &mut
                new_grid_bot);
        }
    }
```

Listing 2.15: grid_bot.rs

```
        pub fn internal_create_bot(&mut self,
                                   base_price: Price,
                                   quote_price: Price,
                                   user: &AccountId,
                                   slippage: u16,
                                   entry_price: &U256C,
                                   pair: &Pair,
                                   grid_bot: &mut GridBot) -> bool {
        if self.status != GridStatus::Running {
           self.internal_create_bot_refund(&user, &pair, PAUSE_OR_SHUTDOWN);
           return false;
        }
        // require!(self.internal_check_oracle_price(*entry_price, base_price.clone(), quote_price.
            clone(), slippage), INVALID_PRICE);
        if !self.internal_check_oracle_price(*entry_price, base_price.clone(), quote_price.clone(),
             slippage) {
           self.internal_create_bot_refund(user, pair, INVALID_PRICE);
           return false;
        }
        // check balance
        // require!(self.internal_get_user_balance(user, &(pair.base_token)) >= base_amount_sell,
        LESS_BASE_TOKEN);
    if self.internal_get_user_balance(user, &(pair.base_token)) < grid_bot.total_base_amount {
        self.internal_create_bot_refund(user, pair, LESS_BASE_TOKEN);
        return false;
    }
    // require!(self.internal_get_user_balance(user, &(pair.quote_token)) >= quote_amount_buy,
        LESS_QUOTE_TOKEN);
    if self.internal_get_user_balance(user, &(pair.quote_token)) < grid_bot.total_quote_amount
        {
        self.internal_create_bot_refund(user, pair, 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);


    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());
    return true;
}
```

Listing 2.16: grid_bot_internal.sol

```
pub fn internal_check_oracle_price(&self, entry_price: U256C, base_price: Price, quote_price:
    Price, slippage: u16) -> bool {
    if base_price.publish_time as u64 * 1000 + self.oracle_valid_time.clone() < env::
        block_timestamp_ms() {
         return false;
    }
    if quote_price.publish_time as u64 * 1000 + self.oracle_valid_time.clone() < env::
            block_timestamp_ms() {
           return false;
        }
        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();


        if entry_price.as_u128() >= oracle_pair_price {
           return (entry_price.as_u128() - oracle_pair_price) * SLIPPAGE_DENOMINATOR as u128 /
               entry_price.as_u128() <= slippage as u128;
        } else {
           return (oracle_pair_price - entry_price.as_u128()) * SLIPPAGE_DENOMINATOR as u128 /
               entry_price.as_u128() <= slippage as u128;
        }
    }
```

Listing 2.17: grid_bot_check.sol

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()`.

```
    pub fn create_bot(&mut self, name: String, pair_id: String, slippage: u16, grid_type: GridType
        ,
                      grid_rate: u16, grid_offset: U128, first_base_amount: U128, first_quote_amount:
                           U128,
                      last_base_amount: U128, last_quote_amount: U128, fill_base_or_quote: bool,
                          grid_sell_count: u16, grid_buy_count: u16,
                      trigger_price: U128, take_profit_price: U128, stop_loss_price: U128,
                          valid_until_time: U128,
                      entry_price: U128) {
        let grid_offset_256 = U256C::from(grid_offset.0);
        let first_base_amount_256 = U256C::from(first_base_amount.0);
        let first_quote_amount_256 = U256C::from(first_quote_amount.0);
        let last_base_amount_256 = U256C::from(last_base_amount.0);
        let last_quote_amount_256 = U256C::from(last_quote_amount.0);
        let trigger_price_256 = U256C::from(trigger_price.0);
        let take_profit_price_256 = U256C::from(take_profit_price.0);
        let stop_loss_price_256 = U256C::from(stop_loss_price.0);
        let valid_until_time_256 = U256C::from(valid_until_time.0);
        let entry_price_256 = U256C::from(entry_price.0);


        require!(self.pair_map.contains_key(&pair_id), INVALID_PAIR_ID);
        let pair = self.pair_map.get(&pair_id).unwrap().clone();
        let user = env::predecessor_account_id();


        // require!(self.status == GridStatus::Running, PAUSE_OR_SHUTDOWN);
        if self.status != GridStatus::Running {
            self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
            return;
        }


        if grid_buy_count + grid_sell_count > MAX_GRID_COUNT {
            self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
            return;
        }


        // calculate all assets
        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(),
                                                   grid_type.clone(), grid_rate.clone(),
                                                       grid_offset_256.clone(), fill_base_or_quote.
                                                       clone());


        // require!(env::attached_deposit() >= STORAGE_FEE, LESS_STORAGE_FEE);
        if !self.internal_check_near_amount(&user, &pair, env::attached_deposit(), base_amount_sell
            , quote_amount_buy) {
            self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
            return;
        }
        // last_quote_amount / last_base_amount > first_quote_amount > first_base_amount
        // amount must u128, u128 * u128 <= u256, so, it's ok
        let (result, reason) = self.internal_check_bot_amount(grid_sell_count, grid_buy_count,
            first_base_amount_256, first_quote_amount_256,
                                                           last_base_amount_256, last_quote_amount_256,
                                                                &user, &pair, base_amount_sell,
                                                               quote_amount_buy);
        if !result {
            self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(), &
                reason);
            return;
        }


        // create bot
        let mut new_grid_bot = GridBot {name, active: false, user: user.clone(), bot_id: "".
            to_string(), closed: false, pair_id, grid_type,
            grid_sell_count: grid_sell_count.clone(), grid_buy_count: grid_buy_count.clone(),
                grid_rate, grid_offset: grid_offset_256,
            first_base_amount: first_base_amount_256, first_quote_amount: first_quote_amount_256,
                last_base_amount: last_base_amount_256,
            last_quote_amount: last_quote_amount_256, fill_base_or_quote, trigger_price:
                trigger_price_256, trigger_price_above_or_below: false,
            take_profit_price: take_profit_price_256, stop_loss_price: stop_loss_price_256,
                valid_until_time: valid_until_time_256,
            total_quote_amount: quote_amount_buy, total_base_amount: base_amount_sell, revenue:
                U256C::from(0), total_revenue: U256C::from(0)
        };


        if self.internal_need_wrap_near(&user, &pair, base_amount_sell, quote_amount_buy) {
            // wrap near to wnear first
            self.deposit_near_to_get_wnear_for_create_bot(&pair, &user, slippage, &entry_price_256,
                 &mut new_grid_bot, env::attached_deposit() - STORAGE_FEE);
        } else {
            // request token price
            self.get_price_for_create_bot(&pair, &user, slippage, &entry_price_256, &mut
                new_grid_bot);
        }
    }
```

Listing 2.18: grid_bot.sol

```
    pub fn get_price_for_create_bot(
        &mut self,
        pair: &Pair,
        user: &AccountId,
        slippage: u16,
        entry_price: &U256C,
        grid_bot: &mut GridBot,
    ) {
        let (promise, tokens) = self.private_create_pair_price_request(pair);
        promise.then(
            Self::ext(env::current_account_id())
                .with_static_gas(GAS_FOR_CREATE_BOT_AFTER_ORACLE)
                .get_price_for_create_bot_callback(tokens.len(), tokens, user, slippage,
                    entry_price, pair, grid_bot),
        );
    }
```

Listing 2.19: oracle.rs

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

Listing 2.20: oracle.rs

```rust
    pub fn internal_create_bot(&mut self,
                               base_price: Price,
                               quote_price: Price,
                               user: &AccountId,
                               slippage: u16,
                               entry_price: &U256C,
                               pair: &Pair,
                               grid_bot: &mut GridBot) -> bool {
    if self.status != GridStatus::Running {
        self.internal_create_bot_refund(&user, &pair, PAUSE_OR_SHUTDOWN);
        return false;
    }
    // require!(self.internal_check_oracle_price(*entry_price, base_price.clone(), quote_price.
        clone(), slippage), INVALID_PRICE);
    if !self.internal_check_oracle_price(*entry_price, base_price.clone(), quote_price.clone(),
         slippage) {
        self.internal_create_bot_refund(user, pair, INVALID_PRICE);
        return false;
    }
    // check balance
    // require!(self.internal_get_user_balance(user, &(pair.base_token)) >= base_amount_sell,
        LESS_BASE_TOKEN);
    if self.internal_get_user_balance(user, &(pair.base_token)) < grid_bot.total_base_amount {
        self.internal_create_bot_refund(user, pair, LESS_BASE_TOKEN);
        return false;
    }
    // require!(self.internal_get_user_balance(user, &(pair.quote_token)) >= quote_amount_buy,
        LESS_QUOTE_TOKEN);
    if self.internal_get_user_balance(user, &(pair.quote_token)) < grid_bot.total_quote_amount
        {
        self.internal_create_bot_refund(user, pair, 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);


        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());
        return true;
    }
```

Listing 2.21: grid_bot_internal.rs

```
    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 {
        let promise_success = is_promise_success();
        if !promise_success.clone() {
            // refund token and near
            self.internal_create_bot_refund_with_near(user, pair, amount + STORAGE_FEE,
                WRAP_TO_WNEAR_ERROR);
            emit::wrap_near_error(user, 0, amount, true);
        } else {
            // deposit
            if !self.internal_deposit(&user.clone(), &self.wnear.clone(), U128::from(amount)) {
                // maybe just need hande one token, but it's ok, no problem
                self.internal_create_bot_refund(user, pair, WRAP_TO_WNEAR_ERROR);
                emit::wrap_near_error(user, 0, amount, true);
            } else {
                // request price
                self.get_price_for_create_bot(pair, user, slippage, entry_price, grid_bot);
            }
        }
        promise_success
    }
```

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.

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

Listing 2.23: grid_bot.rs

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

Listing 2.24: grid_bot_asset.rs

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


pub fn internal_add_refer_recommend_user(&mut self, user: &AccountId, recommender: &AccountId)
     {
    if !self.refer_recommender_user_map.contains_key(recommender) {
        let key = user.to_string() + ":ref_users";
        self.refer_recommender_user_map.insert(recommender, &Vector::new(key.as_bytes().to_vec
            ()));
    }
    let mut ref_users = self.refer_recommender_user_map.get(recommender).unwrap();
    ref_users.push(user);


    self.refer_recommender_user_map.insert(recommender, &ref_users);
}
```

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.

```rust
    pub fn create_bot(&mut self, name: String, pair_id: String, slippage: u16, grid_type:
        GridType,
                    grid_rate: u16, grid_offset: U128, first_base_amount: U128, first_quote_amount
                        : U128,
                    last_base_amount: U128, last_quote_amount: U128, fill_base_or_quote: bool,
                        grid_sell_count: u16, grid_buy_count: u16,
                    trigger_price: U128, take_profit_price: U128, stop_loss_price: U128,
                        valid_until_time: U128,
                    entry_price: U128) {
    let grid_offset_256 = U256C::from(grid_offset.0);
    let first_base_amount_256 = U256C::from(first_base_amount.0);
    let first_quote_amount_256 = U256C::from(first_quote_amount.0);
    let last_base_amount_256 = U256C::from(last_base_amount.0);
    let last_quote_amount_256 = U256C::from(last_quote_amount.0);
    let trigger_price_256 = U256C::from(trigger_price.0);
    let take_profit_price_256 = U256C::from(take_profit_price.0);
    let stop_loss_price_256 = U256C::from(stop_loss_price.0);
    let valid_until_time_256 = U256C::from(valid_until_time.0);
    let entry_price_256 = U256C::from(entry_price.0);


    require!(self.pair_map.contains_key(&pair_id), INVALID_PAIR_ID);
    let pair = self.pair_map.get(&pair_id).unwrap().clone();
    let user = env::predecessor_account_id();


    // require!(self.status == GridStatus::Running, PAUSE_OR_SHUTDOWN);
    if self.status != GridStatus::Running {
        self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
            PAUSE_OR_SHUTDOWN);
        return;
    }


    if grid_buy_count + grid_sell_count > MAX_GRID_COUNT {
        self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
            PAUSE_OR_SHUTDOWN);
        return;
    }


    // calculate all assets
    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(),
                                                grid_type.clone(), grid_rate.clone(),
                                                    grid_offset_256.clone(), fill_base_or_quote.
                                                    clone());


    // require!(env::attached_deposit() >= STORAGE_FEE, LESS_STORAGE_FEE);
    if !self.internal_check_near_amount(&user, &pair, env::attached_deposit(), base_amount_sell
        , quote_amount_buy) {
        self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
            PAUSE_OR_SHUTDOWN);
        return;
    }
    // last_quote_amount / last_base_amount > first_quote_amount > first_base_amount
    // amount must u128, u128 * u128 <= u256, so, it's ok
    let (result, reason) = self.internal_check_bot_amount(grid_sell_count, grid_buy_count,
        first_base_amount_256, first_quote_amount_256,
                                                    last_base_amount_256, last_quote_amount_256,
                                                         &user, &pair, base_amount_sell,
                                                        quote_amount_buy);
    if !result {
        self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(), &
            reason);
        return;
    }


    // create bot
    let mut new_grid_bot = GridBot {name, active: false, user: user.clone(), bot_id: "".
        to_string(), closed: false, pair_id, grid_type,
        grid_sell_count: grid_sell_count.clone(), grid_buy_count: grid_buy_count.clone(),
            grid_rate, grid_offset: grid_offset_256,
        first_base_amount: first_base_amount_256, first_quote_amount: first_quote_amount_256,
            last_base_amount: last_base_amount_256,
        last_quote_amount: last_quote_amount_256, fill_base_or_quote, trigger_price:
            trigger_price_256, trigger_price_above_or_below: false,
        take_profit_price: take_profit_price_256, stop_loss_price: stop_loss_price_256,
            valid_until_time: valid_until_time_256,
        total_quote_amount: quote_amount_buy, total_base_amount: base_amount_sell, revenue:
            U256C::from(0), total_revenue: U256C::from(0)
    };


    if self.internal_need_wrap_near(&user, &pair, base_amount_sell, quote_amount_buy) {
        // wrap near to wnear first
        self.deposit_near_to_get_wnear_for_create_bot(&pair, &user, slippage, &entry_price_256,
             &mut new_grid_bot, env::attached_deposit() - STORAGE_FEE);
     } else {
        // request token price
        self.get_price_for_create_bot(&pair, &user, slippage, &entry_price_256, &mut
            new_grid_bot);
     }
 }
```

Listing 2.26: grid_bot.rs

```
    pub fn internal_create_bot(&mut self,
                               base_price: Price,
                               quote_price: Price,
                               user: &AccountId,
                               slippage: u16,
                               entry_price: &U256C,
                               pair: &Pair,
                               grid_bot: &mut GridBot) -> bool {
        if self.status != GridStatus::Running {
            self.internal_create_bot_refund(&user, &pair, PAUSE_OR_SHUTDOWN);
            return false;
        }
        // require!(self.internal_check_oracle_price(*entry_price, base_price.clone(), quote_price.
            clone(), slippage), INVALID_PRICE);
        if !self.internal_check_oracle_price(*entry_price, base_price.clone(), quote_price.clone(),
            slippage) {
            self.internal_create_bot_refund(user, pair, INVALID_PRICE);
            return false;
        }
        // check balance
        // require!(self.internal_get_user_balance(user, &(pair.base_token)) >= base_amount_sell,
            LESS_BASE_TOKEN);
        if self.internal_get_user_balance(user, &(pair.base_token)) < grid_bot.total_base_amount {
            self.internal_create_bot_refund(user, pair, LESS_BASE_TOKEN);
            return false;
        }
        // require!(self.internal_get_user_balance(user, &(pair.quote_token)) >= quote_amount_buy,
            LESS_QUOTE_TOKEN);
        if self.internal_get_user_balance(user, &(pair.quote_token)) < grid_bot.total_quote_amount {
            self.internal_create_bot_refund(user, pair, 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);


        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());
        return true;
    }
```

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.

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

```
        let last_base_amount_256 = U256C::from(last_base_amount.0);
        let last_quote_amount_256 = U256C::from(last_quote_amount.0);
        let trigger_price_256 = U256C::from(trigger_price.0);
        let take_profit_price_256 = U256C::from(take_profit_price.0);
        let stop_loss_price_256 = U256C::from(stop_loss_price.0);
        let valid_until_time_256 = U256C::from(valid_until_time.0);
        let entry_price_256 = U256C::from(entry_price.0);


        require!(self.pair_map.contains_key(&pair_id), INVALID_PAIR_ID);
        let pair = self.pair_map.get(&pair_id).unwrap().clone();
        let user = env::predecessor_account_id();


        // require!(self.status == GridStatus::Running, PAUSE_OR_SHUTDOWN);
        if self.status != GridStatus::Running {
            self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(), PAUSE_OR_SHUTDOWN);
            return;
        }


        if grid_buy_count + grid_sell_count > MAX_GRID_COUNT {
            self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(), PAUSE_OR_SHUTDOWN);
            return;
        }


        // calculate all assets
        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(),
                                                        grid_type.clone(), grid_rate.clone(), grid_offset_256.clone(), fill_base_or_quote.clone());


        // require!(env::attached_deposit() >= STORAGE_FEE, LESS_STORAGE_FEE);
        if !self.internal_check_near_amount(&user, &pair, env::attached_deposit(), base_amount_sell, quote_amount_buy) {
            self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(), PAUSE_OR_SHUTDOWN);
            return;
        }
        // last_quote_amount / last_base_amount > first_quote_amount > first_base_amount
        // amount must u128, u128 * u128 <= u256, so, it's ok
        let (result, reason) = self.internal_check_bot_amount(grid_sell_count, grid_buy_count, first_base_amount_256, first_quote_amount_256,
                                                                last_base_amount_256, last_quote_amount_256, &user, &pair, base_amount_sell, quote_amount_buy);
        if !result {
            self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(), &
                reason);
            return;
        }


        // create bot
        let mut new_grid_bot = GridBot {name, active: false, user: user.clone(), bot_id: "".
            to_string(), closed: false, pair_id, grid_type,
            grid_sell_count: grid_sell_count.clone(), grid_buy_count: grid_buy_count.clone(),
                grid_rate, grid_offset: grid_offset_256,
            first_base_amount: first_base_amount_256, first_quote_amount: first_quote_amount_256,
                last_base_amount: last_base_amount_256,
            last_quote_amount: last_quote_amount_256, fill_base_or_quote, trigger_price:
                trigger_price_256, trigger_price_above_or_below: false,
            take_profit_price: take_profit_price_256, stop_loss_price: stop_loss_price_256,
                valid_until_time: valid_until_time_256,
            total_quote_amount: quote_amount_buy, total_base_amount: base_amount_sell, revenue:
                U256C::from(0), total_revenue: U256C::from(0)
        };


        if self.internal_need_wrap_near(&user, &pair, base_amount_sell, quote_amount_buy) {
            // wrap near to wnear first
            self.deposit_near_to_get_wnear_for_create_bot(&pair, &user, slippage, &entry_price_256,
                 &mut new_grid_bot, env::attached_deposit() - STORAGE_FEE);
        } else {
            // request token price
            self.get_price_for_create_bot(&pair, &user, slippage, &entry_price_256, &mut
                new_grid_bot);
        }
    }
```

Listing 2.28: grid_bot.rs

```
    pub fn deposit_near_to_get_wnear_for_create_bot(&mut self, pair: &Pair, user: &AccountId,
        slippage: u16, entry_price: &U256C,
                                     grid_bot: &mut GridBot, amount: u128) {
        ext_wnear::ext(self.wnear.clone())
            .with_attached_deposit(amount)
            // .with_static_gas(GAS_FOR_CREATE_BOT_AFTER_NEAR)
            .near_deposit()
            .then(
            Self::ext(env::current_account_id())
                .with_static_gas(GAS_FOR_CREATE_BOT_AFTER_NEAR)
                .after_wrap_near_for_create_bot(
                    pair,
                    user,
                    slippage,
                    entry_price,
                    grid_bot,
                    amount,
                )
    );
}
```

Listing 2.29: grid_bot.rs

```
pub fn deposit_near_to_get_wnear_for_create_bot(&mut self, pair: &Pair, user: &AccountId,
     slippage: u16, entry_price: &U256C,
                                grid_bot: &mut GridBot, amount: u128) {
    ext_wnear::ext(self.wnear.clone())
        .with_attached_deposit(amount)
        // .with_static_gas(GAS_FOR_CREATE_BOT_AFTER_NEAR)
        .near_deposit()
        .then(
        Self::ext(env::current_account_id())
            .with_static_gas(GAS_FOR_CREATE_BOT_AFTER_NEAR)
            .after_wrap_near_for_create_bot(
                pair,
                user,
                slippage,
                entry_price,
                grid_bot,
                amount,
            )
    );
}
```

Listing 2.30: wnear.rs

```
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 {
    let promise_success = is_promise_success();
    if !promise_success.clone() {
        // refund token and near
        self.internal_create_bot_refund_with_near(user, pair, amount + STORAGE_FEE,
            WRAP_TO_WNEAR_ERROR);
        emit::wrap_near_error(user, 0, amount, true);
    } else {
        // deposit
        if !self.internal_deposit(&user.clone(), &self.wnear.clone(), U128::from(amount)) {
            // maybe just need hande one token, but it's ok, no problem
            self.internal_create_bot_refund(user, pair, WRAP_TO_WNEAR_ERROR);
            emit::wrap_near_error(user, 0, amount, true);
        } else {
            // request price
            self.get_price_for_create_bot(pair, user, slippage, entry_price, grid_bot);
        }
    }
    promise_success
}
```

Listing 2.31: wnear.rs

```
pub fn internal_deposit(&mut self, sender_id: &AccountId, token_in: &AccountId, amount: U128)
     -> bool {
        require!(self.global_balances_map.contains_key(token_in), INVALID_TOKEN);
        // require min deposit
        // require!(amount.clone().0 >= self.deposit_limit_map.get(token_in).unwrap().as_u128(),
            LESS_DEPOSIT_AMOUNT);
        if amount.clone().0 < self.deposit_limit_map.get(token_in).unwrap().as_u128() {
            self.internal_token_refund(sender_id, token_in, LESS_DEPOSIT_AMOUNT);
            emit::deposit_failed(sender_id, amount.clone().0, token_in);
            return false;
        }
        // log!("Deposit user:{}, token:{}, amount:{}", sender_id.clone(), token_in.clone(), amount.
            clone().0);
        // add amount to user
        self.internal_increase_asset(sender_id, token_in, &(U256C::from(amount.clone().0)));
        // add amount to global
        self.internal_increase_global_asset(token_in, &(U256C::from(amount.clone().0)));
        // event
        emit::deposit_success(sender_id, amount.clone().0, token_in);
        return true;
    }
```

Listing 2.32: grid_bot_asset.rs

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

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.

```
    pub fn close_bot(&mut self, bot_id: String) {
        assert_one_yocto();
        require!(self.bot_map.contains_key(&bot_id), BOT_NOT_EXIST);
        let mut bot = self.bot_map.get(&bot_id).unwrap().clone();
        let pair = self.pair_map.get(&bot.pair_id).unwrap().clone();
        // check permission, user self close or take profit or stop loss
        // let user = env::predecessor_account_id();
        require!(env::predecessor_account_id() == bot.user, INVALID_USER);
        // require!(self.internal_check_bot_close_permission(&user, &bot), NO_PERMISSION);


        self.internal_close_bot(&env::predecessor_account_id(), &bot_id, &mut bot, &pair);
    }
```

Listing 2.34: grid_bot.rs

```
pub fn internal_close_bot(&mut self, user: &AccountId, bot_id: &String, bot: &mut GridBot,
     pair: &Pair) {
    // sign closed
    bot.closed = true;


    // harvest revenue, must fist execute, will split revenue from bot's asset
    let (revenue_token, revenue) = self.internal_harvest_revenue(bot, pair);
    // unlock token
    self.internal_transfer_assets_to_unlock(&(bot.user), &(pair.base_token), bot.
        total_base_amount.clone());
    self.internal_transfer_assets_to_unlock(&(bot.user), &(pair.quote_token), bot.
        total_quote_amount.clone());


    // withdraw token
    self.internal_withdraw(&(bot.user), &(pair.base_token), bot.total_base_amount);
    self.internal_withdraw(&(bot.user), &(pair.quote_token), bot.total_quote_amount);
    self.internal_withdraw(&(bot.user), &revenue_token, revenue);
    self.bot_map.insert(bot_id, &bot);


    // send claim event
    if revenue.as_u128() > 0 {
        // claim event
        emit::claim(&user, &(bot.user), bot_id.clone(), &revenue_token, revenue);
    }
    emit::close_bot(user, bot_id.clone());
}
```

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.

```
pub fn internal_init_token(&mut self, token: AccountId, min_deposit: U128) -> U256C {
    if self.global_balances_map.contains_key(&token) {
        return U256C::from(0);
    }
    self.global_balances_map.insert(&token, &U256C::from(0));
    self.protocol_fee_map.insert(&token, &U256C::from(0));
    self.deposit_limit_map.insert(&token, &U256C::from(min_deposit.0));
    self.internal_storage_deposit(&env::current_account_id(), &token, DEFAULT_TOKEN_STORAGE_FEE)
        ;
    return U256C::from(DEFAULT_TOKEN_STORAGE_FEE);
}
```

Listing 2.36: grid_bot_internal.rs

```
pub fn internal_storage_deposit(&mut self, account_id: &AccountId, token_id: &AccountId,
    amount: Balance) -> Promise {
    ext_fungible_token::ext(token_id.clone())
        .with_attached_deposit(amount)
        .with_static_gas(GAS_FOR_FT_TRANSFER)
        .storage_deposit(
            Some(account_id.clone()),
            Some(true),
        ).then(
        Self::ext(env::current_account_id())
            .with_static_gas(GAS_FOR_AFTER_FT_TRANSFER)
            .after_storage_deposit(
                account_id.clone(),
                token_id.clone(),
                amount.into(),
            )
    )
}
```

Listing 2.37: token.rs

```
fn after_storage_deposit(
    &mut self,
    account_id: AccountId,
    token_id: AccountId,
    amount: U128,
) -> bool {
    let promise_success = is_promise_success();
    if !promise_success.clone() {
        emit::storage_deposit_failed(&account_id, amount.clone().0, &token_id);
    } else {
        emit::storage_deposit_succeeded(&account_id, amount.clone().0, &token_id);
    }
    promise_success
}
```

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.

```
    pub fn create_bot(&mut self, name: String, pair_id: String, slippage: u16, grid_type:
        GridType,
                    grid_rate: u16, grid_offset: U128, first_base_amount: U128, first_quote_amount
                        : U128,
                    last_base_amount: U128, last_quote_amount: U128, fill_base_or_quote: bool,
                        grid_sell_count: u16, grid_buy_count: u16,
                    trigger_price: U128, take_profit_price: U128, stop_loss_price: U128,
                        valid_until_time: U128,
                    entry_price: U128) {
    let grid_offset_256 = U256C::from(grid_offset.0);
    let first_base_amount_256 = U256C::from(first_base_amount.0);
    let first_quote_amount_256 = U256C::from(first_quote_amount.0);
    let last_base_amount_256 = U256C::from(last_base_amount.0);
    let last_quote_amount_256 = U256C::from(last_quote_amount.0);
    let trigger_price_256 = U256C::from(trigger_price.0);
    let take_profit_price_256 = U256C::from(take_profit_price.0);
    let stop_loss_price_256 = U256C::from(stop_loss_price.0);
    let valid_until_time_256 = U256C::from(valid_until_time.0);
    let entry_price_256 = U256C::from(entry_price.0);


    require!(self.pair_map.contains_key(&pair_id), INVALID_PAIR_ID);
    let pair = self.pair_map.get(&pair_id).unwrap().clone();
    let user = env::predecessor_account_id();


    // require!(self.status == GridStatus::Running, PAUSE_OR_SHUTDOWN);
    if self.status != GridStatus::Running {
        self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
            PAUSE_OR_SHUTDOWN);
        return;
    }


    if grid_buy_count + grid_sell_count > MAX_GRID_COUNT {
        self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
            PAUSE_OR_SHUTDOWN);
        return;
    }


    // calculate all assets
    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(),
                                                grid_type.clone(), grid_rate.clone(),
                                                    grid_offset_256.clone(), fill_base_or_quote.
                                                    clone());


    // require!(env::attached_deposit() >= STORAGE_FEE, LESS_STORAGE_FEE);
    if !self.internal_check_near_amount(&user, &pair, env::attached_deposit(), base_amount_sell
        , quote_amount_buy) {
        self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
            PAUSE_OR_SHUTDOWN);
        return;
    }
    // last_quote_amount / last_base_amount > first_quote_amount > first_base_amount
    // amount must u128, u128 * u128 <= u256, so, it's ok
    let (result, reason) = self.internal_check_bot_amount(grid_sell_count, grid_buy_count,
        first_base_amount_256, first_quote_amount_256,
                                                   last_base_amount_256, last_quote_amount_256,
                                                        &user, &pair, base_amount_sell,
                                                       quote_amount_buy);
    if !result {
        self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(), &
            reason);
        return;
    }


    // create bot
    let mut new_grid_bot = GridBot {name, active: false, user: user.clone(), bot_id: "".
        to_string(), closed: false, pair_id, grid_type,
        grid_sell_count: grid_sell_count.clone(), grid_buy_count: grid_buy_count.clone(),
            grid_rate, grid_offset: grid_offset_256,
        first_base_amount: first_base_amount_256, first_quote_amount: first_quote_amount_256,
            last_base_amount: last_base_amount_256,
        last_quote_amount: last_quote_amount_256, fill_base_or_quote, trigger_price:
            trigger_price_256, trigger_price_above_or_below: false,
        take_profit_price: take_profit_price_256, stop_loss_price: stop_loss_price_256,
            valid_until_time: valid_until_time_256,
        total_quote_amount: quote_amount_buy, total_base_amount: base_amount_sell, revenue:
            U256C::from(0), total_revenue: U256C::from(0)
    };


    if self.internal_need_wrap_near(&user, &pair, base_amount_sell, quote_amount_buy) {
        // wrap near to wnear first
        self.deposit_near_to_get_wnear_for_create_bot(&pair, &user, slippage, &entry_price_256,
             &mut new_grid_bot, env::attached_deposit() - STORAGE_FEE);
    } else {
        // request token price
        self.get_price_for_create_bot(&pair, &user, slippage, &entry_price_256, &mut
            new_grid_bot);
    }
}
```

Listing 2.39: gird_bot.rs

```
        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,
                                        last_base_amount_256: U256C, last_quote_amount_256: U256C, user:
                                            &AccountId, pair: &Pair, base_amount_sell: U256C,
                                            quote_amount_buy: U256C) -> (bool, String) {
        if grid_sell_count > 0 && grid_buy_count > 0 {
            // require!(last_quote_amount_256 * first_base_amount_256 > first_quote_amount_256 *
                last_base_amount_256 , INVALID_FIRST_OR_LAST_AMOUNT);
            if last_quote_amount_256 * first_base_amount_256 <= first_quote_amount_256 *
                last_base_amount_256 {
                return (false, INVALID_FIRST_OR_LAST_AMOUNT.to_string());
            }
        }
        if grid_sell_count > 0 {
            // require!(first_base_amount_256.as_u128() > 0 && first_quote_amount_256.as_u128() >
                0, INVALID_FIRST_OR_LAST_AMOUNT);
            // if first_base_amount_256.as_u128() == 0 || first_quote_amount_256.as_u128() == 0 {
            if last_base_amount_256.as_u128() == 0 || last_quote_amount_256.as_u128() == 0 {
                return (false, INVALID_FIRST_OR_LAST_AMOUNT.to_string());
            }
            // 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);
            if (base_amount_sell.as_u128() / grid_sell_count as u128) < self.deposit_limit_map.get
                (&pair.base_token).unwrap().as_u128() {
                return (false, BASE_TOO_SMALL.to_string());
            }
        }
        if grid_buy_count > 0 {
            // require!(last_base_amount_256.as_u128() > 0 && last_quote_amount_256.as_u128() > 0,
                INVALID_FIRST_OR_LAST_AMOUNT);
            // if last_base_amount_256.as_u128() == 0 || last_quote_amount_256.as_u128() == 0 {
            if first_base_amount_256.as_u128() == 0 || first_quote_amount_256.as_u128() == 0 {
                return (false, INVALID_FIRST_OR_LAST_AMOUNT.to_string());
            }
            // 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);
            if (quote_amount_buy.as_u128() / grid_buy_count as u128) < self.deposit_limit_map.get(&
                pair.quote_token).unwrap().as_u128() {
                self.internal_create_bot_refund(&user, &pair, QUOTE_TOO_SMALL);
                return (false, QUOTE_TOO_SMALL.to_string());
            }
        }
        return (true, "".to_string());
    }
```

Listing 2.40: gird_bot_check.rs

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

Listing 2.41: gird_bot_asset.rs

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

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.

```
pub fn internal_check_order_match(maker_order: Order, taker_order: Order) {
   require!(maker_order.token_buy == taker_order.token_sell, INVALID_ORDER_TOKEN);
   require!(maker_order.token_sell == taker_order.token_buy, INVALID_ORDER_TOKEN);
   require!(taker_order.token_sell != taker_order.token_buy, INVALID_ORDER_TOKEN);
   // taker price and maker price match
   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);
```

```
    }
```

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.

```
    pub fn create_bot(&mut self, name: String, pair_id: String, slippage: u16, grid_type:
        GridType,
                    grid_rate: u16, grid_offset: U128, first_base_amount: U128, first_quote_amount
                        : U128,
                    last_base_amount: U128, last_quote_amount: U128, fill_base_or_quote: bool,
                        grid_sell_count: u16, grid_buy_count: u16,
                    trigger_price: U128, take_profit_price: U128, stop_loss_price: U128,
                        valid_until_time: U128,
                    entry_price: U128) {
    let grid_offset_256 = U256C::from(grid_offset.0);
    let first_base_amount_256 = U256C::from(first_base_amount.0);
    let first_quote_amount_256 = U256C::from(first_quote_amount.0);
    let last_base_amount_256 = U256C::from(last_base_amount.0);
    let last_quote_amount_256 = U256C::from(last_quote_amount.0);
    let trigger_price_256 = U256C::from(trigger_price.0);
    let take_profit_price_256 = U256C::from(take_profit_price.0);
    let stop_loss_price_256 = U256C::from(stop_loss_price.0);
    let valid_until_time_256 = U256C::from(valid_until_time.0);
    let entry_price_256 = U256C::from(entry_price.0);


    require!(self.pair_map.contains_key(&pair_id), INVALID_PAIR_ID);
    let pair = self.pair_map.get(&pair_id).unwrap().clone();
    let user = env::predecessor_account_id();


    // require!(self.status == GridStatus::Running, PAUSE_OR_SHUTDOWN);
    if self.status != GridStatus::Running {
        self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
            PAUSE_OR_SHUTDOWN);
        return;
    }
```

```


        if grid_buy_count + grid_sell_count > MAX_GRID_COUNT {
            self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(), PAUSE_OR_SHUTDOWN);
            return;
        }


        // calculate all assets
        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(),
                                                       grid_type.clone(), grid_rate.clone(), grid_offset_256.clone(), fill_base_or_quote.clone());


        // require!(env::attached_deposit() >= STORAGE_FEE, LESS_STORAGE_FEE);
        if !self.internal_check_near_amount(&user, &pair, env::attached_deposit(), base_amount_sell, quote_amount_buy) {
            self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(), PAUSE_OR_SHUTDOWN);
            return;
        }
        // last_quote_amount / last_base_amount > first_quote_amount > first_base_amount
        // amount must u128, u128 * u128 <= u256, so, it's ok
        let (result, reason) = self.internal_check_bot_amount(grid_sell_count, grid_buy_count, first_base_amount_256, first_quote_amount_256,
                                                            last_base_amount_256, last_quote_amount_256, &user, &pair, base_amount_sell, quote_amount_buy);
        if !result {
            self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(), &reason);
            return;
        }


        // create bot
        let mut new_grid_bot = GridBot {name, active: false, user: user.clone(), bot_id: "".to_string(), closed: false, pair_id, grid_type,
            grid_sell_count: grid_sell_count.clone(), grid_buy_count: grid_buy_count.clone(), grid_rate, grid_offset: grid_offset_256,
            first_base_amount: first_base_amount_256, first_quote_amount: first_quote_amount_256, last_base_amount: last_base_amount_256,
            last_quote_amount: last_quote_amount_256, fill_base_or_quote, trigger_price: trigger_price_256, trigger_price_above_or_below: false,
            take_profit_price: take_profit_price_256, stop_loss_price: stop_loss_price_256, valid_until_time: valid_until_time_256,
            total_quote_amount: quote_amount_buy, total_base_amount: base_amount_sell, revenue: U256C::from(0), total_revenue: U256C::from(0)
        };


        if self.internal_need_wrap_near(&user, &pair, base_amount_sell, quote_amount_buy) {
            // wrap near to wnear first
            self.deposit_near_to_get_wnear_for_create_bot(&pair, &user, slippage, &entry_price_256,
                 &mut new_grid_bot, env::attached_deposit() - STORAGE_FEE);
        } else {
            // request token price
            self.get_price_for_create_bot(&pair, &user, slippage, &entry_price_256, &mut
                new_grid_bot);
        }
    }
```

Listing 2.44: gird_bot.rs

```
        pub fn internal_create_bot(&mut self,
                                   base_price: Price,
                                   quote_price: Price,
                                   user: &AccountId,
                                   slippage: u16,
                                   entry_price: &U256C,
                                   pair: &Pair,
                                   grid_bot: &mut GridBot) -> bool {
        if self.status != GridStatus::Running {
            self.internal_create_bot_refund(&user, &pair, PAUSE_OR_SHUTDOWN);
            return false;
        }
        // require!(self.internal_check_oracle_price(*entry_price, base_price.clone(), quote_price.
            clone(), slippage), INVALID_PRICE);
        if !self.internal_check_oracle_price(*entry_price, base_price.clone(), quote_price.clone(),
             slippage) {
            self.internal_create_bot_refund(user, pair, INVALID_PRICE);
            return false;
        }
        // check balance
        // require!(self.internal_get_user_balance(user, &(pair.base_token)) >= base_amount_sell,
            LESS_BASE_TOKEN);
        if self.internal_get_user_balance(user, &(pair.base_token)) < grid_bot.total_base_amount {
            self.internal_create_bot_refund(user, pair, LESS_BASE_TOKEN);
            return false;
        }
        // require!(self.internal_get_user_balance(user, &(pair.quote_token)) >= quote_amount_buy,
            LESS_QUOTE_TOKEN);
        if self.internal_get_user_balance(user, &(pair.quote_token)) < grid_bot.total_quote_amount
            {
            self.internal_create_bot_refund(user, pair, 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);


        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());
        return true;
    }
```

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.

```
    pub fn create_bot(&mut self, name: String, pair_id: String, slippage: u16, grid_type: GridType
        ,
                      grid_rate: u16, grid_offset: U128, first_base_amount: U128, first_quote_amount
                          : U128,
                      last_base_amount: U128, last_quote_amount: U128, fill_base_or_quote: bool,
                          grid_sell_count: u16, grid_buy_count: u16,
                    trigger_price: U128, take_profit_price: U128, stop_loss_price: U128,
                        valid_until_time: U128,
                    entry_price: U128) {
        let grid_offset_256 = U256C::from(grid_offset.0);
        let first_base_amount_256 = U256C::from(first_base_amount.0);
        let first_quote_amount_256 = U256C::from(first_quote_amount.0);
        let last_base_amount_256 = U256C::from(last_base_amount.0);
        let last_quote_amount_256 = U256C::from(last_quote_amount.0);
        let trigger_price_256 = U256C::from(trigger_price.0);
        let take_profit_price_256 = U256C::from(take_profit_price.0);
        let stop_loss_price_256 = U256C::from(stop_loss_price.0);
        let valid_until_time_256 = U256C::from(valid_until_time.0);
        let entry_price_256 = U256C::from(entry_price.0);


        require!(self.pair_map.contains_key(&pair_id), INVALID_PAIR_ID);
        let pair = self.pair_map.get(&pair_id).unwrap().clone();
        let user = env::predecessor_account_id();


        // require!(self.status == GridStatus::Running, PAUSE_OR_SHUTDOWN);
        if self.status != GridStatus::Running {
            self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
            return;
        }


        if grid_buy_count + grid_sell_count > MAX_GRID_COUNT {
            self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
            return;
        }


        // calculate all assets
        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(),
                                                  grid_type.clone(), grid_rate.clone(),
                                                      grid_offset_256.clone(), fill_base_or_quote.
                                                      clone());


        // require!(env::attached_deposit() >= STORAGE_FEE, LESS_STORAGE_FEE);
        if !self.internal_check_near_amount(&user, &pair, env::attached_deposit(), base_amount_sell
            , quote_amount_buy) {
            self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(),
                PAUSE_OR_SHUTDOWN);
            return;
        }
        // last_quote_amount / last_base_amount > first_quote_amount > first_base_amount
        // amount must u128, u128 * u128 <= u256, so, it's ok
```

```
    let (result, reason) = self.internal_check_bot_amount(grid_sell_count, grid_buy_count,
        first_base_amount_256, first_quote_amount_256,
                                                           last_base_amount_256, last_quote_amount_256,
                                                                &user, &pair, base_amount_sell,
                                                               quote_amount_buy);
    if !result {
        self.internal_create_bot_refund_with_near(&user, &pair, env::attached_deposit(), &
            reason);
        return;
    }


    // create bot
    let mut new_grid_bot = GridBot {name, active: false, user: user.clone(), bot_id: "".
        to_string(), closed: false, pair_id, grid_type,
        grid_sell_count: grid_sell_count.clone(), grid_buy_count: grid_buy_count.clone(),
            grid_rate, grid_offset: grid_offset_256,
        first_base_amount: first_base_amount_256, first_quote_amount: first_quote_amount_256,
            last_base_amount: last_base_amount_256,
        last_quote_amount: last_quote_amount_256, fill_base_or_quote, trigger_price:
            trigger_price_256, trigger_price_above_or_below: false,
        take_profit_price: take_profit_price_256, stop_loss_price: stop_loss_price_256,
            valid_until_time: valid_until_time_256,
        total_quote_amount: quote_amount_buy, total_base_amount: base_amount_sell, revenue:
            U256C::from(0), total_revenue: U256C::from(0)
    };


    if self.internal_need_wrap_near(&user, &pair, base_amount_sell, quote_amount_buy) {
        // wrap near to wnear first
        self.deposit_near_to_get_wnear_for_create_bot(&pair, &user, slippage, &entry_price_256,
             &mut new_grid_bot, env::attached_deposit() - STORAGE_FEE);
    } else {
        // request token price
        self.get_price_for_create_bot(&pair, &user, slippage, &entry_price_256, &mut
            new_grid_bot);
    }
}
```

Listing 2.46: grid_bot.rs

```
pub fn internal_check_near_amount(&mut self, user: &AccountId, pair: &Pair, near_amount: u128,
     base_amount_sell: U256C, quote_amount_buy: U256C) -> bool {
    if pair.quote_token != self.wnear && pair.base_token != self.wnear && near_amount !=
        STORAGE_FEE {
        return false;
    }
    let wnear_balance = self.internal_get_user_balance(&user, &self.wnear);
    if pair.base_token == self.wnear {
        if wnear_balance.as_u128() >= base_amount_sell.as_u128() && near_amount != STORAGE_FEE
            {
            // wnear balance is enough, but user support near
            return false;
            }
            if wnear_balance.as_u128() < base_amount_sell.as_u128() && near_amount != (base_amount_sell.as_u128() + STORAGE_FEE) { //audit
                // wnear balance is not enough, but near is less
                return false;
            }
        }
        if pair.quote_token == self.wnear {
            if wnear_balance.as_u128() >= quote_amount_buy.as_u128() && near_amount != STORAGE_FEE {
                // wnear balance is enough, but user support near
                return false;
            }
            if wnear_balance.as_u128() < quote_amount_buy.as_u128() && near_amount != (quote_amount_buy.as_u128() + STORAGE_FEE) {
                // wnear balance is not enough, but near is less
                return false;
            }
        }
        // if wnear not register, will revert, it's ok
        let wnear_min_deposit = self.deposit_limit_map.get(&self.wnear).unwrap();
        if pair.base_token == self.wnear && base_amount_sell.as_u128() < wnear_min_deposit.as_u128()
            || pair.quote_token == self.wnear && quote_amount_buy.as_u128() < wnear_min_deposit.as_u128() {
            return false;
        }
        return true;
    }
```

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.

```rust
    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) {
        let bot = self.bot_map.get(&bot_id.clone()).unwrap().clone();
        let pair = self.pair_map.get(&bot.pair_id).unwrap().clone();
        let (maker_order, in_orderbook) = self.query_order(bot_id.clone(), forward_or_reverse, level
            );
        // matching check
        GridBotContract::internal_check_order_match(maker_order.clone(), taker_order.clone());


        // calculate
        let (taker_sell, taker_buy, current_filled, made_order) = GridBotContract::
            internal_calculate_matching(maker_order.clone(), taker_order.clone(), took_sell,
            took_buy);


        // place into orderbook
        if !in_orderbook {
            self.internal_place_order(bot_id.clone(), maker_order.clone(), forward_or_reverse.clone
                (), level.clone());
        }
        // update filled
        let maker_order = self.internal_update_order_filled(bot_id.clone(), forward_or_reverse.clone
            (), level.clone(), current_filled.clone());
        emit::order_update(bot_id.clone(), forward_or_reverse.clone(), level.clone(), &maker_order);


        // place opposite order
        let opposite_order = GridBotContract::internal_get_opposite_order(&made_order, bot.clone(),
            forward_or_reverse.clone(), level.clone());
        self.internal_place_order(bot_id.clone(), opposite_order.clone(), !forward_or_reverse.clone
            (), level.clone());


        // query real_opposite_order
        let (real_opposite_order, _) = self.query_order(bot_id.clone(), !forward_or_reverse.clone(),
             level.clone());
        emit::order_update(bot_id.clone(), !forward_or_reverse.clone(), level.clone(), &
            real_opposite_order);


        // calculate bot's revenue
        let (revenue_token, revenue, maker_fee) = self.internal_calculate_bot_revenue(
            forward_or_reverse.clone(), maker_order.clone(), opposite_order, current_filled.clone()
            );


        // add revenue
        // let bot_mut = self.bot_map.get_mut(&bot_id.clone()).unwrap();
        let mut bot = self.bot_map.get(&bot_id.clone()).unwrap();
        bot.revenue += revenue;
        bot.total_revenue += revenue;
        // update bot asset
        GridBotContract::internal_update_bot_asset(&mut bot, &pair, taker_order.token_buy.clone(),
            taker_buy.as_u128(), taker_sell.as_u128());


        // bot asset transfer
        self.internal_reduce_locked_assets(&(bot.user), &(taker_order.token_buy), &taker_buy);
        self.internal_increase_locked_assets(&(bot.user), &(taker_order.token_sell), &taker_sell);


        // allocate refer fee
        let (protocol_fee, _) = self.internal_allocate_refer_fee(&maker_fee, &bot.user, &
            revenue_token);
        // handle protocol fee
        self.internal_add_protocol_fee_from_revenue(&mut bot, &revenue_token, maker_fee,
            protocol_fee, &pair);


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


        // 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);
        return (taker_sell, taker_buy, bot.user.clone(), maker_fee, revenue, bot.revenue, bot.
            total_revenue);
    }
```

Listing 2.48: orderbook_internal.sol

```
pub fn internal_calculate_bot_revenue(&self, forward_or_reverse: bool, order: Order,
    opposite_order: Order, current_filled: U256C) -> (AccountId, U256C, U256C) {
    if forward_or_reverse {
        return (opposite_order.token_sell, U256C::from(0), U256C::from(0));
    }
    // let forward_order = GridBotContract::internal_get_first_forward_order(bot, pair, level);
    let revenue_token;
    let mut revenue;
    // TODO had made_order, maybe can use mad_order
    // mad_order, opposite_order
    if opposite_order.fill_buy_or_sell {
        // current_filled token is forward_order's buy token
        // revenue token is forward_order's sell token
        let forward_sold = current_filled.clone() * opposite_order.amount_sell / opposite_order
            .amount_buy;
        let reverse_bought = current_filled.clone() * order.amount_buy / order.amount_sell;
        require!(reverse_bought >= forward_sold, INVALID_REVENUE);
        revenue_token = opposite_order.token_sell;
        revenue = reverse_bought - forward_sold;
    } else {
        // current_filled token is forward_order's sell token
        // revenue token is forward_order's buy token
        let forward_bought = current_filled.clone() * opposite_order.amount_buy /
```

```
            opposite_order.amount_sell;
        let reverse_sold = current_filled.clone() * order.amount_sell / order.amount_buy;
        require!(forward_bought >= reverse_sold, INVALID_REVENUE);
        revenue_token = opposite_order.token_buy;
        revenue = forward_bought - reverse_sold;
    };
    let protocol_fee = revenue * U256C::from(self.protocol_fee_rate.clone()) / U256C::from(
        PROTOCOL_FEE_DENOMINATOR);
    revenue -= protocol_fee;
    return (revenue_token, revenue.clone(), protocol_fee.clone());
}


pub fn internal_calculate_taker_fee(&self, took_buy: U256C) -> (U256C, U256C) {
    let taker_fee = took_buy * U256C::from(self.taker_fee_rate.clone()) / U256C::from(
        PROTOCOL_FEE_DENOMINATOR);
    return (took_buy - taker_fee, taker_fee);
}
```

Listing 2.49: orderbook_internal.sol

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`.

2.2 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            }
```

```
            return true
        }
        return true;
    }
```

Listing 2.50: grid_bot_internal.rs

```
    pub fn internal_reduce_asset(&mut self, user: &AccountId, token: &AccountId, amount: &U256C) {
        let mut user_balances = self.user_balances_map.get(user).unwrap_or_else(|| {
            let mut map = LookupMap::new(StorageKey::UserBalanceSubKey(user.clone()));
            map.insert(token, &U256C::from(0));
            map
        });


        let balance = user_balances.get(token).unwrap_or(U256C::from(0));
        user_balances.insert(token, &(balance - amount));


        self.user_balances_map.insert(user, &user_balances);
    }
```

Listing 2.51: grid_bot_asset.rs

```
    pub fn internal_reduce_refer_fee(&mut self, user: &AccountId, token: &AccountId, amount: &U128
        ) {
        if amount.0 == 0 {
            return;
        }
        if !self.refer_fee_map.contains_key(user) {
            self.refer_fee_map.insert(user, &LookupMap::new(StorageKey::ReferFeeSubKey(user.clone()
                )));
        }
        let mut tokens_map = self.refer_fee_map.get(user).unwrap();
        require!(tokens_map.contains_key(token), INVALID_TOKEN);
        tokens_map.insert(token, &U128::from(tokens_map.get(token).unwrap().0 - amount.clone().0));
        self.refer_fee_map.insert(user, &tokens_map);
    }
```

Listing 2.52: 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.

```
pub fn internal_ft_transfer_near(&mut self, receiver_id: &AccountId, amount: Balance,
    effect_global_balance: bool) -> Promise {
   Promise::new(receiver_id.clone()).transfer(amount)
       .then(
       Self::ext(env::current_account_id())
           .with_static_gas(GAS_FOR_AFTER_FT_TRANSFER)
           .after_ft_transfer_near(
               receiver_id.clone(),
               self.wnear.clone(),
               amount.into(),
               effect_global_balance,
           )
       )
}
```

Listing 2.53: token.rs

```
fn after_ft_transfer_near(
   &mut self,
   account_id: AccountId,
   token_id: AccountId,
   amount: U128,
   effect_global_balance: bool,
) -> bool {
   let promise_success = is_promise_success();
   if !promise_success.clone() {
       emit::withdraw_failed(&account_id, amount.clone().0, &token_id);
       if effect_global_balance {
           self.internal_increase_withdraw_near_error_effect_global(&account_id, &amount);
       } else {
           self.internal_increase_withdraw_near_error(&account_id, &amount);
       }
   } else {
       emit::withdraw_succeeded(&account_id, amount.clone().0, &token_id);
       if effect_global_balance {
           // reduce from global asset
           self.internal_reduce_global_asset(&token_id, &(U256C::from(amount.clone().0)))
       }
   }
   promise_success
}
```

Listing 2.54: 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.

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

Listing 2.55: 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.

2.3 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).

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

Listing 2.56: grid_bot.rs

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

Listing 2.57: grid_bot.rs

```
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) {
   require!(env::attached_deposit() == DEFAULT_TOKEN_STORAGE_FEE * 2, LESS_TOKEN_STORAGE_FEE);
   require!(env::predecessor_account_id() == self.owner_id, ERR_NOT_ALLOWED);
   require!(base_token != quote_token, INVALID_TOKEN);
   let pair_key = GridBotContract::internal_get_pair_key(base_token.clone(), quote_token.clone
       ());
   require!(!self.pair_map.contains_key(&pair_key), PAIR_EXIST);
   let pair = Pair{
      base_token: base_token.clone(),
      quote_token: quote_token.clone(),
      base_oracle_id: self.internal_format_price_identifier(base_oracle_id),
      quote_oracle_id: self.internal_format_price_identifier(quote_oracle_id),
   };
   self.pair_map.insert(&pair_key, &pair);
   self.internal_init_token(base_token, base_min_deposit);
   self.internal_init_token(quote_token, quote_min_deposit);
}
```

Listing 2.58: 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.