BEOSIN
Blockchain Security

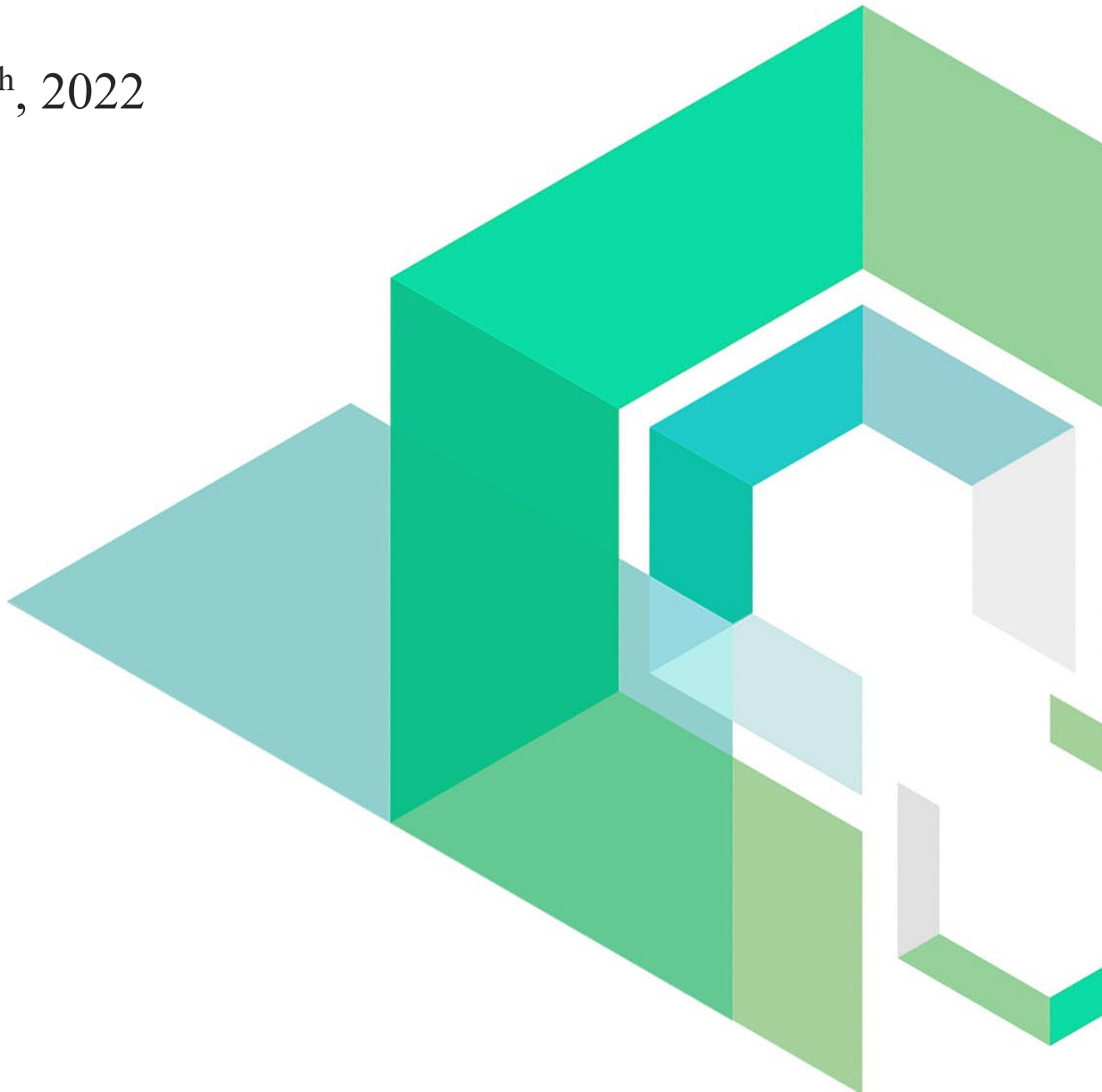
CRAFTING

Smart Contract Security Audit

V1.1

No. 202204141429

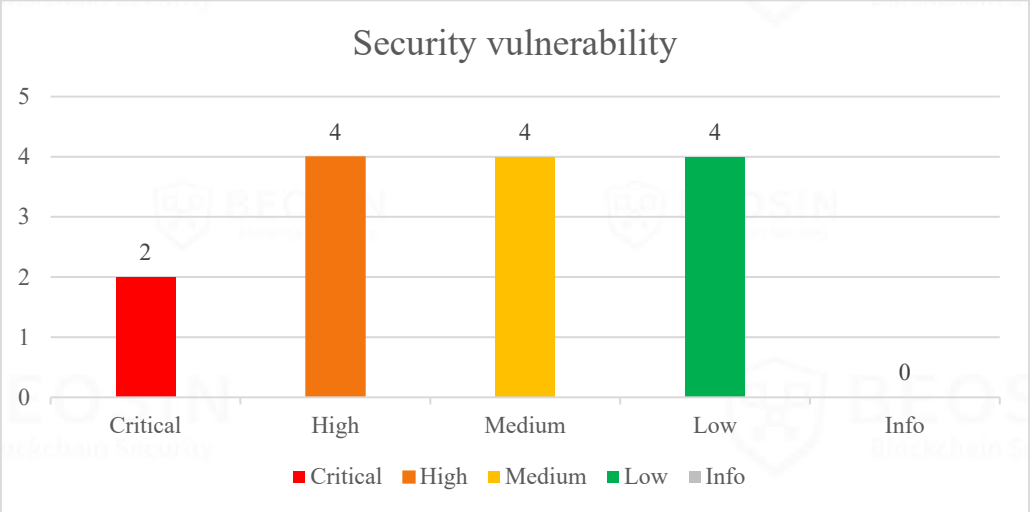
April 14th, 2022



Contents

Summary of Audit Results

After auditing, 2 Critical-risk, 4 High-risk,4 Medium-risk and 4 Low-risk items were identified in the Crafting project. Specific audit details will be presented in the **Findings section**. Users should pay attention to the following aspects when interacting with this project：



*Notes：

- **Risk Description:**

1. According to the project party, the pair price of the tokens queried by the Crafting project is provided by Flux Protocol FPO (First Party Oracle). According to the Flux project, the data comes from some of the most reputable sources Defi. Users should pay attention to Flux Protocol FPO data provider security in real time.

2. According to the project party, rUSD will be used as the settlement token of the Crafting project. The "rUSD/USD" and "USN/USD" pairs price queried in the contract will always be 100,000,000 (1 rUSD = 1 USD, 1 USN = 1 USD), with a precision of 8. The real price of rUSD and USN are maintained externally and may differ from the price returned in the contract. Users should pay attention to the price changes of rUSD and USN in real time.

3. According to the Near blockchain account-contract model, the Crafting main contract deployment account and the Crafting contract have the same permissions. If DAO is not used, the Crafting main contract deployment account will have the possibility of manipulating user assets. Since the DAO contract was not included in this audit, the project party stated that the Crafting contract authority will be transferred to DAO, and users should pay attention to whether the project uses DAO governance in real time.

- **Note for Project Party**

1. The owner of the Crafting contract needs to register an account in the mortgaged token contract, otherwise the interest will be lost during liquidation.

2. When setting the feed pair "token1/token2" for the pledged tokens and Rafting synthetic assets, project party needs to keep making sure that token2 is always consistent, otherwise there will be errors when calculating the amount of different assets.

- **Project Description:**

Crafting is a synthetic asset issuance protocol where users can mint a synthetic asset, such as USD, by staking backed tokens and automatically take a long position on that asset. Users can also exchange minted assets for other assets through crafting contracts, so as to short assets and long other assets. The assets minted by all users correspond to the liabilities of the entire system, and the debt ratio of each user has been determined at the time of forging, so that their respective benefits can be calculated.

1 Overview

1.1 Project Overview

Project Name	Crafting
Platform	Near Blockchain
Audit Scope	https://github.com/crafting-finance/crafting-core-near
Commit Hash	14da2c740fda9aa2e2f8345353beaa8c55a085d5 (original) 30c3db4aad069ca2020ab1cfb35b0ffa2e03fefd (fixed) 2ba5a7a3f8087a66650e07abe3e9c3216bdebaa8 (updated)

1.2 Audit Overview

Audit work duration：February 25, 2022 – April 14, 2022

Update Details: May 11, 2022. Business logic update.

Audit methods: Formal Verification, Static Analysis, Typical Case Testing and Manual Review.

Audit team: Beosin Technology Co. Ltd.

2 Findings

Index	Risk description	Severity level	Status
Crafting-1	The return value of *query_raft_amount* method is abnormal	**Critical**	Fixed
Crafting-2	Multiple methods handle 'Promise' exceptions	**Critical**	Fixed
Crafting-3	The *feed_price* method is not used by the contract	**High**	Fixed
Crafting-4	The *swap_in_accountbook* method updates the debt pool account book abnormally	**High**	Fixed
Crafting-5	The *redeem_in_debtpool* method query exception	**High**	Fixed
Crafting-6	The *liquidate_in_debtpool* method has a conditional judgment defect	**High**	Fixed
Crafting-7	*mint_callback* method visibility exception	**Medium**	Fixed
Crafting-8	*UnorderedMap::get()* usage exception	**Medium**	Fixed
Crafting-9	*Mint* method implementation exception	**Medium**	Fixed
Crafting-10	The *register_account* method design does not conform to the NEP-145	**Medium**	Fixed
Crafting-11	Multiple functions lack return value checks	**Low**	Fixed
Crafting-12	The returned price value has exceptions in *get_price* method	**Low**	Acknowledged
Crafting-13	Centralization risk	**Low**	Acknowledged
Crafting-14	*ft_on_transfer* method conditional judgment is flawed	**Low**	Fixed

Status Notes:

1. Crafting-12 is not fixed, which will cause the price of the "rUSD/USD" and "USN/USD" pairs query to always be 100,000,000 (8-digit precision, which means 1 rUSD = 1 USD, 1 USN = 1 USD), which may deviate from the actual price. And according to the project party, the price is updated through the off-chain script.
2. Crafting-13 is not fixed, and according to the Near blockchain account-contract model, the Crafting main contract deployment account and the Crafting contract have the same permissions. If DAO is not used, the Crafting main contract deployment account will have the possibility of manipulating user assets. Since the DAO contract was not included in this audit, the project party stated that the Crafting contract authority will be transferred to DAO, and users should pay attention to whether the project uses DAO governance in real time.

Risk Details Description:

[Crafting-1] The return value of *query_raft_amount* method is abnormal

Severity Level	**Critical**
Type	General Vulnerability
Lines	/crafting/src/debtpool.rs #L27-36
Description	The WrappedBalance.amount returned by calling the *query_raft_amount* method is always 0, which will cause errors in the balance calculation of multiple functions in the crafting contract.

```
pub(crate) fn query_raft_amount(&self, raft_id: &AccountId) -> WrappedBalance {
    let opt_wbalance = self.raft_amounts.get(raft_id);
    if opt_wbalance.is_some() {
        opt_wbalance.unwrap();
    }

    return WrappedBalance{
        amount: 0,
        is_positive: true,
    }
}
```

Figure 1 *query_raft_amount* method (unfixed)

Recommendations	It is recommended to modify the balance return logic.
Status	Fixed.

```
pub(crate) fn query_raft_amount(&self, raft_id: &AccountId) -> i128 {
    if let Some(amount) = self.raft_amounts.get(raft_id) {
        amount
    } else {
        0
    }
}
```

Figure 2 query_raft_amount method (fixed)

[Crafting-2] Multiple methods handle 'Promise' exceptions

Severity Level	**Critical**
Type	General Vulnerability
Lines	/crafting/src/accountbook.rs #L76, L83
Description	*account_book_callback_deposit* and *account_book_callback_withdraw* methods do not judge the returned 'Promise' and directly update the account book, which will cause the user to update the account book even if fails to call method between contracts.

```
#[private]
pub fn account_book_callback_deposit(&mut self, sender_id: AccountId, raft_id: AccountId,
                                     amount: Balance, raft_amount: Balance, user_raft_amount: Balance) {
    self.account_book.insert_raft_amount(&raft_id, raft_amount + amount);
    self.account_book.insert_user_raft_amount(&sender_id, &raft_id, user_raft_amount + amount);
}

#[private]
pub fn account_book_callback_withdraw(&mut self, sender_id: AccountId, raft_id: AccountId,
                                      amount: Balance, raft_amount: Balance, user_raft_amount: Balance) {
    self.account_book.insert_raft_amount(&raft_id, raft_amount - amount);
    self.account_book.insert_user_raft_amount(&sender_id, &raft_id, user_raft_amount - amount);
}
```

Figure 3 *account_book_callback_deposit & account_book_callback_withdraw* method (unfixed)

Recommendations	It is recommended to use 'PromiseResult' to judge the returned 'Promise'.
Status	Fixed.

```
#[private]
pub fn account_book_callback_deposit(&mut self, sender_id: AccountId, raft_id: AccountId, amount: i128) {
    match env::promise_result(0) {
        PromiseResult::NotReady => unreachable!(),
        PromiseResult::Successful(_value) => {
            let raft_amount = self.account_book.query_raft_amount(&raft_id);
            let user_raft_amount = self.account_book.query_user_raft_amount(&sender_id, &raft_id);

            self.account_book.insert_raft_amount(&raft_id, raft_amount + amount);
            self.account_book.insert_user_raft_amount(&sender_id, &raft_id, user_raft_amount + amount);

            log!("{} deposited {} {}", sender_id, amount, raft_id);
        }
        PromiseResult::Failed => panic!("ERR_FAILED_RAFT_DEPOSIT"),
    };
}

#[private]
pub fn account_book_callback_withdraw(&mut self, sender_id: AccountId, raft_id: AccountId, amount: i128) {
    match env::promise_result(0) {
        PromiseResult::NotReady => unreachable!(),
        PromiseResult::Successful(_value) => {
            let raft_amount = self.account_book.query_raft_amount(&raft_id);
            let user_raft_amount = self.account_book.query_user_raft_amount(&sender_id, &raft_id);

            self.account_book.insert_raft_amount(&raft_id, raft_amount - amount);
            self.account_book.insert_user_raft_amount(&sender_id, &raft_id, user_raft_amount - amount);

            log!("{} withdrew {} {}", sender_id, amount, raft_id);
        }
        PromiseResult::Failed => panic!("ERR_FAILED_RAFT_WITHDRAW"),
    };
}
```

Figure 4 *account_book_callback_deposit & account_book_callback_withdraw* method (fixed)

[Crafting-3] The *feed_price* method is not used by the contract

Severity Level	**High**
Type	Business Security
Lines	/crafting/src/oracle.rs #L26-28
Description	The *feed_price* method in oracle.rs is not used by the crafting contract and will result in prices not being set, which will make some contract functions that call the *get_price* method panic.

```
pub fn feed_price(&mut self, asset: &AccountId, price: u128) {
    self.prices.insert(asset, &price);
}
```

Figure 5 *feed_price* method (unfixed)

```
pub fn get_price(&self, asset: &AccountId) -> u128 {
    let opt = self.prices.get(asset);
    assert!(opt.is_some());
    opt.unwrap()
}
```

Figure 6 *get_price* method (unfixed)

Recommendations	It is recommended to add a method to set the price in the crafting contract.
Status	Fixed. The project has changed the get_price method to the get_price_entry method to request and store the pair price from Flux Protocol FPO.

```
pub fn get_price_entry(&self, pair: String, provider: AccountId) -> Promise {
    fpo::get_entry(
        pair.clone(),
        provider.clone(),
        self.oracle.clone(),
        NO_DEPOSIT,
        env::prepaid_gas() - GAS_FOR_FT_TRANSFER_CALL,
    ).then(
        ext_self::set_price_entry(
            pair.clone(),
            provider.clone(),
            env::current_account_id(),
            NO_DEPOSIT,
            GAS_FOR_RESOLVE_TRANSFER,
        )
    )
}
}
```

Figure 7 *get_price_entry* method

[Crafting-4] The *swap_in_accountbook* method updates the debt pool account book abnormally

Severity Level	**High**
Type	Business Security
Lines	/crafting/src/lib.rs #L293
Description	In the *swap_in accountbook* method, when updating the old pool ledger, 'the new_swap_amount' was incorrectly updated into the old pool.

```
pub fn swap_in_accountbook(&mut self, old_raft_id: AccountId, new_raft_id: AccountId, swap_amount: Balance) {
    self.assert_contract_running();

    assert!(self.is_in_whitelisted_rafts(&old_raft_id), "{}", errors::RAFT_NOT_WHITELISTED);
    assert!(self.is_in_whitelisted_rafts(&new_raft_id), "{}", errors::RAFT_NOT_WHITELISTED);
    assert!(swap_amount > 0, "{}", errors::INVALID_RAFT_AMOUNT);

    let sender_id = env::predecessor_account_id();

    let old_raft_amount = self.account_book.query_raft_amount(&old_raft_id);
    assert!(old_raft_amount >= swap_amount, "{}", errors::INVALID_RAFT_AMOUNT);
    let old_user_raft_amount = self.account_book.query_user_raft_amount(&sender_id, &old_raft_id);
    assert!(old_user_raft_amount >= swap_amount, "{}", errors::INVALID_RAFT_AMOUNT);

    // charge transaction fee
    let exchange_fee_amount = swap_amount * self.exchange_fee as u128 / utils::FEE_DIVISOR as u128;
    let owner_raft_amount = self.account_book.query_user_raft_amount(&self.owner_id, &old_raft_id);
    self.account_book.insert_user_raft_amount(&self.owner_id, &old_raft_id, owner_raft_amount + exchange_fee_amount);

    // processing in the account book
    self.account_book.insert_raft_amount(&old_raft_id, old_raft_amount - swap_amount + exchange_fee_amount);
    self.account_book.insert_user_raft_amount(&sender_id, &old_raft_id, old_user_raft_amount - swap_amount);

    let new_swap_amount = self.price_oracle.get_price(&old_raft_id) * (swap_amount - exchange_fee_amount)
        / self.price_oracle.get_price(&new_raft_id);
    let new_raft_amount = self.account_book.query_raft_amount(&new_raft_id);
    self.account_book.insert_raft_amount(&new_raft_id, new_raft_amount + new_swap_amount);

    let new_user_raft_amount = self.account_book.query_user_raft_amount(&sender_id, &new_raft_id);
    self.account_book.insert_user_raft_amount(&sender_id, &new_raft_id, new_user_raft_amount + new_swap_amount);

    // processing in the debt pool
    let old_raft_amount = self.debt_pool.query_raft_amount(&old_raft_id);
    self.debt_pool.calc_sub_raft_amount(&old_raft_id, &old_raft_amount, new_swap_amount);

    let new_raft_amount = self.debt_pool.query_raft_amount(&new_raft_id);
    self.debt_pool.calc_add_raft_amount(&new_raft_id, &new_raft_amount, new_swap_amount);
}
```

Figure 8 *swap_in_accountbook* method (unfixed)

Recommendations	It is recommended to change 'new_swap_amount' to 'old_raft_amount - swap_amount.0 + exchange_fee_amount'.
Status	Fixed.

```
    // processing in the debt pool
    let owner_raft_amount = self.debt_pool.query_user_raft_amount(&self.owner_id, &old_raft_id);
    self.debt_pool.insert_user_raft_amount(&self.owner_id, &old_raft_id, owner_raft_amount + exchange_fee_amount);

    let old_raft_amount = self.debt_pool.query_raft_amount(&old_raft_id);
    self.debt_pool.insert_raft_amount(&old_raft_id, old_raft_amount - swap_amount.0 + exchange_fee_amount);

    let new_raft_amount = self.debt_pool.query_raft_amount(&new_raft_id);
    self.debt_pool.insert_raft_amount(&new_raft_id, new_raft_amount + new_swap_amount);

    log!("{} exchanged {} {} for {} {}. In the debt pool: {}",
        sender_id, swap_amount.0, old_raft_id, new_swap_amount, new_raft_id, false);
}
```

Figure 9 *swap_in_accountbook* method (fixed)

[Crafting-5] The *redeem_in_debtpool* method query exception

Severity Level	**High**
Type	Business Security
Lines	/crafting/src/lib.rs #L258-L260
Description	Since the *mint* method of the main contract does not process contract.user_collaterals, it will cause the *redeem_in_debtpool* method to panic when querying the collateral id.

```rust
#[payable]
pub fn mint(&mut self, token: ValidAccountId, raft: ValidAccountId, raft_amount: Balance, join_debtpool: bool) {
    assert_one_yocto();
    self.assert_contract_running();

    assert!(self.is_in_whitelisted_tokens(token.as_ref()));
    assert!(self.is_in_whitelisted_rafts(raft.as_ref()));

    let token_amount = env::attached_deposit();
    assert!(token_amount > 0, "{}", errors::NoAttachedDeposit);
    assert!(raft_amount > 0, "{}", errors::SyntheticAmountError);

    let caller = env::predecessor_account_id();
    if join_debtpool {
        let token_decimals = self.query_token(token.as_ref()).unwrap().decimals;
        let raft_decimals = self.query_raft(raft.as_ref()).unwrap().decimals;

        let leverage_ratio = (self.price_oracle.get_price(raft.as_ref()) * raft_amount * 10u128.pow(token_decimals))
            / (self.price_oracle.get_price(token.as_ref()) * token_amount * 10u128.pow(raft_decimals));

        let (min, max) = self.leverage_ratio;
        assert!(leverage_ratio >= min.into());
        assert!(leverage_ratio <= max.into());

        self.debt_pool.join(&self.price_oracle, &caller, raft.as_ref(), raft_amount);
    } else {
        let token_asset = self.query_token(token.as_ref()).unwrap();
        let raft_asset = self.query_token(raft.as_ref()).unwrap();

        let token_decimals = token_asset.decimals;
        let raft_decimals = raft_asset.decimals;

        let collateral_ratio = (self.price_oracle.get_price(token.as_ref()) * token_amount * 10u128.pow(raft_decimals) * 100)
            / (self.price_oracle.get_price(raft.as_ref()) * raft_amount * 10u128.pow(token_decimals));

        assert!(collateral_ratio >= token_asset.collateral_ratio);

        self.account_book.mint(&caller, raft.as_ref(), raft_amount);
    }

    let collateral = Collateral {
        issuer: caller,
        token: token.as_ref().clone(),
        token_amount,
        raft: raft.as_ref().clone(),
        raft_amount,
        join_debtpool,
        block_index: env::block_index(),
        create_time: env::block_timestamp(),
        state: 0,
    };

    self.collaterals.push(&collateral);
}
```

Figure 10 *mint* method (unfixed)

Recommendations	It is recommended to record the user's collateral id in the *mint* method in the main

	contract, and add a user's collateral query method.
Status	Fixed.

```
        let collateral = Collateral {
            issuer: sender_id.clone(),
            token_id: token_id.clone(),
            token_amount,
            raft_id: raft_id.clone(),
            raft_amount,
            join_debtpool,
            block_index: env::block_height(),
            create_time: env::block_timestamp(),
            state: 0,
        };

        self.collaterals.push(&collateral);

        let opt_collateral_ids: Option<Vector<CollateralId>> = self.user_collaterals.get(&sender_id);
        let mut collateral_ids;
        if opt_collateral_ids.is_some() {
            collateral_ids = opt_collateral_ids.unwrap();
        } else {
            collateral_ids = Vector::new(StorageKey::UserCollateralId);
        }
        collateral_ids.push(&(self.collaterals.len() - 1));
        self.user_collaterals.insert(&sender_id, &collateral_ids);

        log!("{} pledged {} {} to mint {} {}. In the debt pool: {}",
            sender_id, token_amount, token_id, raft_amount, raft_id, join_debtpool);
    }
```

Figure 11 *mint* method (fixed)

[Crafting-6] The *liquidate_in_debtpool* method has a conditional judgment defect

Severity Level	**High**
Type	Business Security
Lines	/crafting/src/owner.rs #L225-L263
Description	In the *liquidate_in_debtpool* method, the case of "net_debt<0" is not judged, which will cause the tokens stored in the contract to be withdrawn to the account of the liquidator.

```
pub fn liquidate_in_debtpool(&mut self, user: AccountId, liquidation: AccountId) -> PromiseOrValue<U128> {
    assert_one_yocto();
    self.assert_owner();
    self.assert_account_liquidating(&user);

    let collateral_ids: Option<Vector<CollateralId>> = self.user_collaterals.get(&user);
    assert!(collateral_ids.is_some(), "{}", errors::NO_COLLATERAL);

    // calculate the value of user debt
    let user_debt_ratio = self.debt_pool.query_debt_ratio(&user);
    let raft_total_value = self.calc_debtpool_total_value();
    let user_debt = raft_total_value * user_debt_ratio / utils::RATIO_DIVISOR;

    let mut net_debt = user_debt - self.calc_debtpool_user_total_value(&user);

    // destroy all synthetic assets held by the user
    for (raft, amount) in self.debt_pool.query_user_raft_amounts(&user).iter() {
        let raft_amount = self.debt_pool.query_raft_amount(raft);
        self.debt_pool.insert_raft_amount(raft, raft_amount - *amount);
        self.debt_pool.remove_user_raft_amount(&user, raft);
    }

    // remove user debt ratio
    self.debt_pool.remove_debt_ratio(&user);

    // recalculating debt ratio
    let new_raft_total_value = self.calc_debtpool_total_value();
    self.debt_pool.calc_all_debt_ratio(raft_total_value, new_raft_total_value);

    // return of collateral assets
    for collateral_id in collateral_ids.unwrap().iter() {
        let opt_collateral = self.query_collateral(collateral_id);
        if opt_collateral.is_none() { continue; }
        let mut collateral = opt_collateral.unwrap();
        if collateral.issuer != user { continue; }
        if collateral.join_debtpool == false { continue; }
        if collateral.state != 0 { continue; }

        if net_debt == 0 {
            let mut account = self.internal_unwrap_account(&user);
            account.withdraw(&collateral.token_id, collateral.token_amount as u128);
            self.internal_save_account(&user, account);
            self.internal_send_tokens(&user, &collateral.token_id, collateral.token_amount as u128);
        } else {
            let token_asset = self.query_token(&collateral.token_id).unwrap();
            let token_entry = self.oracle_requester.get_price_entry(&token_asset.feed_pair, &token_asset.feed_provider);
            let token_value = (
                collateral.token_amount *
                    token_entry.price.0 as i128 *
                    utils::PRICE_UNIT *
                    utils::AMOUNT_UNIT) / (
                10i128.pow(token_asset.decimals) *
                    10i128.pow(token_entry.decimals as u32));

            if token_value >= net_debt {
                let net_debt_amount = (
                    net_debt *
                        10i128.pow(token_asset.decimals) *
                        10i128.pow(token_entry.decimals as u32)) / (
                    token_entry.price.0 as i128 *
                        utils::AMOUNT_UNIT *
                        utils::PRICE_UNIT);

                net_debt = 0;

                let mut account = self.internal_unwrap_account(&user);
                account.withdraw(&collateral.token_id, collateral.token_amount as u128);
                self.internal_save_account(&user, account);
                self.internal_send_tokens(&liquidation, &collateral.token_id, net_debt_amount as u128);
                self.internal_send_tokens(&user, &collateral.token_id, (collateral.token_amount - net_debt_amount) as u128);
            } else {
                net_debt = net_debt - token_value;

                let mut account = self.internal_unwrap_account(&user);
                account.withdraw(&collateral.token_id, collateral.token_amount as u128);
                self.internal_save_account(&user, account);
                self.internal_send_tokens(&liquidation, &collateral.token_id, collateral.token_amount as u128);
            }

            // update collateral state
            collateral.state = 1;
            self.collaterals.replace(collateral_id, &collateral);
        }
    }
```

Figure 12 *liquidate_in_debtpool* method (unfixed)

Recommendations	It is recommended to add the processing logic for liquidating user assets when "net debt<0".

Status Fixed.

```rust
pub fn liquidate_in_debtpool(&mut self, user: AccountId, liquidation: AccountId) -> PromiseOrValue<U128> {
    assert_one_yocto();
    self.assert_owner_or_operators();
    self.assert_account_liquidating(&user);

    let opt_collateral_ids: Option<UnorderedSet<CollateralId>> = self.user_collaterals.get(&user);
    require!(opt_collateral_ids.is_some(), errors::NO_COLLATERAL);
    let collateral_ids: UnorderedSet<CollateralId> = opt_collateral_ids.unwrap();

    // calculate the value of user debt
    let raft_total_value = self.calc_debtpool_total_value();
    let user_debt_value = self.calc_debtpool_user_debt_value(&user, raft_total_value);

    let mut net_debt = user_debt_value - self.calc_debtpool_user_total_value(&user);
    if net_debt < 0 {
        self.account_states.insert(&user, &AccountState::Running);
        log!("Profitable status, no liquidation. user: {}", user);
        return PromiseOrValue::Value(U128(0));
    }

    // destroy all synthetic assets held by the user
    for (raft, amount) in self.debt_pool.query_user_raft_amounts(&user).iter() {
        let raft_amount = self.debt_pool.query_raft_amount(raft);
        self.debt_pool.insert_raft_amount(raft, raft_amount - *amount);
        self.debt_pool.remove_user_raft_amount(&user, raft);
    }

    // remove user debt ratio
    self.debt_pool.remove_debt_ratio(&user);

    // recalculating debt ratio
    let new_raft_total_value = self.calc_debtpool_total_value();
    self.debt_pool.calc_all_debt_ratio(raft_total_value, new_raft_total_value);

    // return of collateral assets
    let mut vec: Vec<CollateralId> = Vec::new();
    for collateral_id in collateral_ids.iter() {
        let opt_collateral = self.query_collateral(collateral_id);
        if opt_collateral.is_none() { continue; }
        let collateral = opt_collateral.unwrap();
        if collateral.issuer != user { continue; }
        if collateral.join_debtpool == false { continue; }

        if net_debt == 0 {
            let mut account = self.internal_unwrap_account(&user);
            account.withdraw(&collateral.token_id, collateral.token_amount.0 as u128);
            self.internal_save_account(&user, account);
            self.internal_send_tokens(&user, &collateral.token_id, collateral.token_amount.0 as u128);
        } else {
            let token_asset = self.query_token(&collateral.token_id).unwrap();
            let token_entry = self.get_price(&token_asset.feed_pair);

            let token_value: i128 = (
                collateral.token_amount.0 *
                    token_entry.price.0 as i128
            ) / (
                10i128.pow(token_asset.decimals + token_entry.decimals as u32 - utils::PRICE_DECIMALS - utils::AMOUNT_DECIMALS)
            );

            if token_value >= net_debt {
                let net_debt_amount = (
                    net_debt *
                        10i128.pow(token_asset.decimals + token_entry.decimals as u32 - utils::PRICE_DECIMALS - utils::AMOUNT_DECIMALS)
                ) / (
                    token_entry.price.0 as i128
                );

                net_debt = 0;

                let mut account = self.internal_unwrap_account(&user);
                account.withdraw(&collateral.token_id, collateral.token_amount.0 as u128);
                self.internal_save_account(&user, account);
                self.internal_send_tokens(&liquidation, &collateral.token_id, net_debt_amount as u128);
                self.internal_send_tokens(&user, &collateral.token_id, (collateral.token_amount.0 - net_debt_amount) as u128);
            } else {
                net_debt = net_debt - token_value;

                let mut account = self.internal_unwrap_account(&user);
                account.withdraw(&collateral.token_id, collateral.token_amount.0 as u128);
                self.internal_save_account(&user, account);
                self.internal_send_tokens(&liquidation, &collateral.token_id, collateral.token_amount.0 as u128);
            }
        }

        // invalid CollateralId to be removed
        vec.push(collateral_id);
    }

    self.remove_user_collaterals(&user, collateral_ids, vec);

    self.account_states.insert(&user, &AccountState::Running);
    log!("Liquidating from {} to {} by {}", user, liquidation, env::predecessor_account_id());
    PromiseOrValue::Value(U128(0))
}
```

Figure 13 *liquidate_in_debtpool* method (fixed)

[Crafting-7] *mint_callback* method visibility exception

Severity Level	Medium
Type	Coding Conventions
Lines	/crafting/src/lib.rs #L186
Description	The *mint_callback* method has incorrect visibility, which will cause the *mint* method cannot be call back correctly.

```
#[private]
fn mint_callback(&mut self, sender_id: AccountId, token_id: AccountId, token_amount: Balance,
                 raft_id: AccountId, raft_amount: Balance, join_debtpool: bool) {
    if join_debtpool {
        let token_decimals = self.query_token(&token_id).unwrap().decimals;
        let raft_decimals = self.query_raft(&raft_id).unwrap().decimals;

        let leverage_ratio = (self.price_oracle.get_price(&raft_id) * raft_amount * 10u128.pow(token_decimals))
            / (self.price_oracle.get_price(&token_id) * token_amount * 10u128.pow(raft_decimals));

        let (min, max) = self.leverage_ratio;
        assert!(leverage_ratio >= min.into(), "{}", errors::INVALID_LEVERAGE_RATIO);
        assert!(leverage_ratio <= max.into(), "{}", errors::INVALID_LEVERAGE_RATIO);

        self.debt_pool.join(&self.price_oracle, &sender_id, &raft_id, raft_amount);
    } else {
        let token_asset = self.query_token(&token_id).unwrap();
        let raft_asset = self.query_token(&raft_id).unwrap();

        let token_decimals = token_asset.decimals;
        let raft_decimals = raft_asset.decimals;

        let collateral_ratio = (self.price_oracle.get_price(&token_id) * token_amount * 10u128.pow(raft_decimals) * 100)
            / (self.price_oracle.get_price(&raft_id) * raft_amount * 10u128.pow(token_decimals));

        assert!(collateral_ratio >= token_asset.collateral_ratio, "{}", errors::INVALID_COLLATERAL_RATIO);

        self.account_book.mint(&sender_id, &raft_id, raft_amount);
    }

    let collateral = Collateral {
        issuer: sender_id,
        token_id: token_id.clone(),
        token_amount,
        raft_id: raft_id.clone(),
        raft_amount,
        join_debtpool,
        block_index: env::block_height(),
        create_time: env::block_timestamp(),
        state: 0,
    };

    self.collaterals.push(&collateral);
}
```

Figure 14 *mint_callback* method (unfixed)

Recommendations	It is recommended to change the *mint_callback* method visibility to public (pub fn).
Status	Fixed. The *mint_callback* function has been removed.

[Crafting-8] *UnorderedMap::get()* usage exception

Severity Level	Medium
Type	Business Security
Lines	/crafting/src/accountbook.rs #L24 /crafting/src/debtpool.rs #L59
Description	The *query_raft_amount* method in debtpool and accountbook mods use the *Unordered_Map::get()* method incorrectly. When the key passed to the *Unordered_Map::get()* method does not exist, the contract will panic.

```
pub(crate) fn query_raft_amount(&self, raft_id: &AccountId) -> WrappedBalance {
    let opt_wbalance = self.raft_amounts.get(raft_id);
    if opt_wbalance.is_some() {
        return opt_wbalance.unwrap();
    }

    WrappedBalance {
        amount: 0,
        is_positive: true,
    }
}
```

Figure 15 *query_raft_amount* method in debtpool mod (unfixed)

```
pub(crate) fn query_raft_amount(&self, raft_id: &AccountId) -> Balance {
    self.raft_amounts.get(raft_id).unwrap_or(0)
}
```

Figure 16 *query_raft_amount* method in accountbook mod (unfixed)

Recommendations	It is recommended to use the Some() method to capture the return value.
Status	Fixed.

```
pub(crate) fn query_raft_amount(&self, raft_id: &AccountId) -> i128 {
    if let Some(amount) = self.raft_amounts.get(raft_id) {
        amount
    } else {
        0
    }
}
```

Figure 17 query_raft_amount method in debtpool and accountbook mods (fixed)

[Crafting-9] *mint* method implementation exception

Severity Level	Medium
Type	Business Security
Lines	/raft-token/src/lib.rs #L77
Description	The *mint* method will call the *internal_register_account* method to register the account, but the repeated registration of the same account will cause the contract to panic, which will make the *withdraw_in_accountbook* method in the Crafting contract unable to mint craft-tokens for the user multiple times.

```
pub fn mint(&mut self, account_id: AccountId, amount: U128) {
    assert_eq!(env::predecessor_account_id(), self.owner_id, "Unauthorized");

    self.token.internal_register_account(&account_id);
    self.token.internal_deposit(&account_id, amount.into());
}
```

Figure 18 *mint* method (unfixed)

```
pub fn internal_register_account(&mut self, account_id: &AccountId) {
    if self.accounts.insert(account_id, &0).is_some() {
        env::panic_str("The account is already registered");
    }
}
```

Figure 19 *internal_register_account* method

```
#[payable]
pub fn withdraw_in_accountbook(&mut self, raft_id: AccountId, amount: Balance) -> Promise {
    assert_one_yocto();
    self.assert_contract_running();

    assert!(amount > 0, "{}", errors::ILLEGAL_WITHDRAW_AMOUNT);

    let sender_id = env::predecessor_account_id();
    let raft_amount = self.account_book.query_raft_amount(&raft_id);
    let user_raft_amount = self.account_book.query_user_raft_amount(&sender_id, &raft_id);
    assert!(raft_amount >= amount, "{}", errors::INVALID_RAFT_AMOUNT);
    assert!(user_raft_amount >= amount, "{}", errors::INVALID_USER_RAFT_AMOUNT);

    ext_enhanced_fungible_token::mint(
        sender_id.clone(),
        U128(amount),
        raft_id.clone(),
        utils::ONE_YOCTO,
        utils::GAS_FOR_FT_TRANSFER,
    ).then(ext_self::account_book_callback_withdraw(
        sender_id.clone(),
        raft_id.clone(),
        amount,
        raft_amount,
        user_raft_amount,
        env::current_account_id(),
        utils::NO_DEPOSIT,
        utils::GAS_FOR_FT_TRANSFER,
    ))
}
```

Figure 20 *withdraw_in_accountbookt* method in contract

Recommendations	It is recommended to check whether there is an already registered account before registering an account in the *mint* method.
Status	Fixed.

```rust
pub fn mint(&mut self, account_id: AccountId, amount: U128) {
    assert_eq!(env::predecessor_account_id(), self.owner_id, "Unauthorized");

    if !self.token.accounts.contains_key(&account_id) {
        self.token.internal_register_account(&account_id);
        log!("Add new user: {}", account_id);
    }

    self.token.internal_deposit(&account_id, amount.into());
    log!("{} minted {} {}", account_id, amount.0, self.ft_metadata().symbol);
    FtMint {
        owner_id: &account_id,
        amount: &amount,
        memo: Some("withdraw"),
    }.emit();
}
```

Figure 21 *mint* method (fixed)

[Crafting-10] The *register_account* method design does not conform to the NEP-145

Severity Level	**Medium**
Type	Coding Conventions
Lines	/raft-token/src/lib.rs #L81-L86
Description	The design of the *register_account* method violates the Near blockchain storage staking principle. Attackers can register many accounts increase to huge storage costs for contracts.

```
pub fn register_account(&mut self, account_id: AccountId) {
    if !self.accounts.contains(&account_id) {
        self.accounts.insert(&account_id);
        self.token.internal_register_account(&account_id);
    }
}
```

Figure 22 *register_account* method

Recommendations	It is recommended to delete the *register_account* method and use *storage_deposit* method to register an account.
Status	Fixed. The *register_account method* has been removed

[Crafting-11] Multiple functions lack return value checks

Severity Level	**Low**
Type	Business Security
Lines	/crafting/src/views.rs #L70, L87, L119, L136
Description	In the view.rs source file, none of the *debtpool_raft_amount, debtpool_user_raft_amount, accountbook_user_raft_amount,* and *accountbook_raft_amount* methods check the return value of the *is_in_whitelisted_rafts* method.

```
fn is_in_whitelisted_rafts(&self, raft_id: &AccountId) -> bool {
    if self.whitelisted_rafts.contains(raft_id) {
        return true;
    }

    false
}
```

Figure 23 *is_in_whitelisted_rafts* method

```
pub fn debtpool_raft_amount(&self, raft_id: AccountId) -> WrappedBalance {
    self.is_in_whitelisted_rafts(&raft_id);

    self.debt_pool.query_raft_amount(&raft_id)
}
```

Figure 24 *debtpool_raft_amount* method (unfixed)

```
pub fn debtpool_user_raft_amount(&self, user: AccountId, raft_id: AccountId) -> Balance {
    self.assert_query_authority(user.clone());
    self.is_in_whitelisted_rafts(&raft_id);

    self.debt_pool.query_user_raft_amount(&user, &raft_id)
}
```

Figure 25 *debtpool_user_raft_amount* method (unfixed)

```
pub fn accountbook_raft_amount(&self, raft_id: AccountId) -> Balance {
    self.is_in_whitelisted_rafts(&raft_id);

    self.account_book.query_raft_amount(&raft_id)
}
```

Figure 26 *accountbook_user_raft_amount* method (unfixed)

Recommendations	It is recommended to check the return value of *is_in_whitelisted_rafts* method in the *debtpool_raft_amount, debtpool_user_raft_amount, accountbook_user_raft_amount,* and *accountbook_raft_amount* methods.
Status	Fixed.

```
pub fn debtpool_raft_amount(&self, raft_id: AccountId) -> i128 {
    assert!(self.is_in_whitelisted_rafts(&raft_id), "{}", errors::RAFT_NOT_WHITELISTED);

    self.debt_pool.query_raft_amount(&raft_id)
}
```

Figure 27 *debtpool_raft_amount* method (fixed)

```
pub fn debtpool_user_raft_amount(&self, user: AccountId, raft_id: AccountId) -> i128 {
    assert!(self.is_in_whitelisted_rafts(&raft_id), "{}", errors::RAFT_NOT_WHITELISTED);
    self.debt_pool.query_user_raft_amount(&user, &raft_id)
}
```

Figure 28 *debtpool_raft_amount* method (fixed)

```
pub fn accountbook_user_raft_amount(&self, user: AccountId, raft_id: AccountId) -> i128 {
    assert!(self.is_in_whitelisted_rafts(&raft_id), "{}", errors::RAFT_NOT_WHITELISTED);
    self.account_book.query_user_raft_amount(&user, &raft_id)
}
```

Figure 29 *debtpool_raft_amount* method (fixed)

```
pub fn accountbook_raft_amount(&self, raft_id: AccountId) -> i128 {
    assert!(self.is_in_whitelisted_rafts(&raft_id), "{}", errors::RAFT_NOT_WHITELISTED);

    self.account_book.query_raft_amount(&raft_id)
}
```

Figure 30 *debtpool_raft_amount* method (fixed)

[Crafting-12] The returned price value has exceptions in *get_price* method

Severity Level	**Low**
Type	Business Security
Lines	/crafting/src/oracle.rs #L80-L93
Description	The *get_price* method will directly return the preset value when the pair is "rUSD/USD" or "USN/USD" instead of querying the price from the oracle, and the price will not be updated in real time.

```
pub(crate) fn get_price(&self, pair: &String) -> PriceEntry {
    if pair == "rUSD / USD" || pair == "USN / USD" {
        return PriceEntry {
            price: U128(10u128.pow(utils::PRICE_DECIMALS)),
            decimals: utils::PRICE_DECIMALS as u16,
            last_update: 0,
        };
    }

    let opt_entry = self.oracle_pairs.get(pair);
    require!(opt_entry.is_some(), errors::PRICE_ENTRY_NOT_EXIST);

    opt_entry.unwrap()
}
```

Figure 31 *get_price* method

Recommendations	It is recommended to query the price in real time from the Oracle.
Status	Acknowledged. According to the project party, the project party will call the *get_price_entry* method in the contract every minute through the off-chain script to request and save the price from the Flux Protocol FPO, and the project party will use rUSD as the USD settlement token. The price of the 'rUSD/USD' and 'USN/USD' pairs are maintained externally.

[Crafting-13] Centralization risk

Severity Level	**Low**
Type	Business Security
Lines	None
Description	According to the account-contract model of the near blockchain, the deployment account of the Crafting main contract can also call *mint* method or call *burn* method to operate the user's Rafting-token, even upgrade the Crafting main contract.
Recommendations	It is recommended to use DAO contract to manage and upgrade the Crafting main contract.
Status	Acknowledged. According to the project party, Crafting contract will be governed by DAO.

[Crafting-14] *ft_on_transfer* method conditional judgment is flawed

Severity Level	**Low**
Type	Business Security
Lines	/crafting/src/debtpool.rs #L27-36
Description	When the msg received by *ft_on_transfer* is empty, the user's token will be directly deposited into the contract, and the user will not be able to directly withdraw the token deposited into the contract by mistake.

```
fn ft_on_transfer(
    &mut self,
    sender_id: AccountId,
    amount: U128,
    msg: String,
) -> PromiseOrValue<U128> {
    self.assert_contract_running();
    self.assert_account_running(&sender_id);

    let token_id = env::predecessor_account_id();
    if !msg.is_empty() {
        let message: TokenReceiverMessage = serde_json::from_str(&msg).expect(errors::ILLEGAL_MSG_FORMAT);
        self.mint(&sender_id, &token_id, amount.0 as i128, &message.raft_id,
                  message.raft_amount.0, message.join_debtpool);
    }

    let account = self.internal_unwrap_or_default_account(&sender_id);
    self.internal_save_account(&sender_id, account);
    self.internal_deposit(&sender_id, &token_id, amount.into());

    PromiseOrValue::Value(U128(0))
}
```

Figure 32 *ft_on_transfer* method (unfixed)

Recommendations	It is recommended to throw panic when *ft_on_transfer* method receives an empty msg.
Status	Fixed.

```
fn ft_on_transfer(
    &mut self,
    sender_id: AccountId,
    amount: U128,
    msg: String,
) -> PromiseOrValue<U128> {
    self.assert_contract_running();
    self.assert_account_running(&sender_id);
    assert!(!msg.is_empty(), "{}", errors::INVALID_PARAMETER);

    let token_id = env::predecessor_account_id();
    let message: TokenReceiverMessage = serde_json::from_str(&msg).expect(errors::ILLEGAL_MSG_FORMAT);
    self.mint(&sender_id, &token_id, amount.0 as i128, &message.raft_id,
              message.raft_amount.0, message.join_debtpool);

    let account = self.internal_unwrap_or_default_account(&sender_id);
    self.internal_save_account(&sender_id, account);
    self.internal_deposit(&sender_id, &token_id, amount.into());

    PromiseOrValue::Value(U128(0))
}
```

Figure 33 *ft_on_transfer* method (fixed)

3 Appendix

3.1 Vulnerability Assessment Metrics and Status in Smart Contracts

3.1.1 Metrics

In order to objectively assess the severity level of vulnerabilities in blockchain systems, this report provides detailed assessment metrics for security vulnerabilities in smart contracts with reference to CVSS 3.1 (Common Vulnerability Scoring System Ver 3.1).

According to the severity level of vulnerability, the vulnerabilities are classified into four levels: "critical", "high", "medium" and "low". It mainly relies on the degree of impact and likelihood of exploitation of the vulnerability, supplemented by other comprehensive factors to determine of the severity level.

Likelihood \ Impact	Severe	High	Medium	Low
Probable	Critical	High	Medium	Low
Possible	High	High	Medium	Low
Unlikely	Medium	Medium	Low	Info
Rare	Low	Low	Info	Info

3.1.2 Degree of impact

- **Severe**

Severe impact generally refers to the vulnerability can have a serious impact on the confidentiality, integrity, availability of smart contracts or their economic model, which can cause substantial economic losses to the contract business system, large-scale data disruption, loss of authority management, failure of key functions, loss of credibility, or indirectly affect the operation of other smart contracts associated with it and cause substantial losses, as well as other severe and mostly irreversible harm.

- **High**

High impact generally refers to the vulnerability can have a relatively serious impact on the confidentiality, integrity, availability of the smart contract or its economic model, which can cause a greater economic loss, local functional unavailability, loss of credibility and other impact to the contract business system.

- **Medium**

Medium impact generally refers to the vulnerability can have a relatively minor impact on the confidentiality, integrity, availability of the smart contract or its economic model, which can cause a small amount of economic loss to the contract business system, individual business unavailability and other impact.

- **Low**

Low impact generally refers to the vulnerability can have a minor impact on the smart contract, which can pose certain security threat to the contract business system and needs to be improved.

3.1.4 Likelihood of Exploitation

- **Probable**

Probable likelihood generally means that the cost required to exploit the vulnerability is low, with no special exploitation threshold, and the vulnerability can be triggered consistently.

- **Possible**

Possible likelihood generally means that exploiting such vulnerability requires a certain cost, or there are certain conditions for exploitation, and the vulnerability is not easily and consistently triggered.

- **Unlikely**

Unlikely likelihood generally means that the vulnerability requires a high cost, or the exploitation conditions are very demanding and the vulnerability is highly difficult to trigger.

- **Rare**

Rare likelihood generally means that the vulnerability requires an extremely high cost or the conditions for exploitation are extremely difficult to achieve.

3.1.5 Fix Results Status

Status	Description
Fixed	The project party fully fixes a vulnerability.
Partially Fixed	The project party did not fully fix the issue, but only mitigated the issue.
Acknowledged	The project party confirms and chooses to ignore the issue.

3.2 Audit Categories

No.	Categories	Subitems
1	Coding Conventions	Deprecated Items
		Redundant Code
		Assert Usage
		Gas Consumption
2	General Vulnerability	Integer Overflow/Underflow
		Reentrancy
		Pseudo-random Number Generator (PRNG)
		Transaction-Ordering Dependence
		DoS (Denial of Service)
		Function Call Permissions
		PromiseResult Security
		Returned Value Security
		Replay Attack
		Third-party Protocol Interface Consistency
3	Business Security	Business Logics
		Business Implementations
		Manipulable Token Price
		Centralized Asset Control
		Asset Tradability
		Arbitrage Attack

Beosin classified the security issues of smart contracts into three categories: Coding Conventions, General Vulnerability, Business Security. Their specific definitions are as follows:

- **Coding Conventions**

Audit whether smart contracts follow recommended language security coding practices. For example, smart contracts developed in Solidity language should fix the compiler version and do not use deprecated keywords.

- **General Vulnerability**

General Vulnerability include some common vulnerabilities that may appear in smart contract projects. These vulnerabilities are mainly related to the characteristics of the smart contract itself, such as integer overflow/underflow and denial of service attacks.

- **Business Security**

Business security is mainly related to some issues related to the business realized by each project, and has a relatively strong pertinence. For example, whether the lock-up plan in the code match the white paper, or the flash loan attack caused by the incorrect setting of the price acquisition oracle.

*Note that the project may suffer stake losses due to the integrated third-party protocol. This is not something Beosin can control. Business security requires the participation of the project party. The project party and users need to stay vigilant at all times.

3.3 Disclaimer

The Audit Report issued by Beosin is related to the services agreed in the relevant service agreement. The Project Party or the Served Party (hereinafter referred to as the "Served Party") can only be used within the conditions and scope agreed in the service agreement. Other third parties shall not transmit, disclose, quote, rely on or tamper with the Audit Report issued for any purpose.

The Audit Report issued by Beosin is made solely for the code, and any description, expression or wording contained therein shall not be interpreted as affirmation or confirmation of the project, nor shall any warranty or guarantee be given as to the absolute flawlessness of the code analyzed, the code team, the business model or legal compliance.

The Audit Report issued by Beosin is only based on the code provided by the Served Party and the technology currently available to Beosin. However, due to the technical limitations of any organization, and in the event that the code provided by the Served Party is missing information, tampered with, deleted, hidden or subsequently altered, the audit report may still fail to fully enumerate all the risks.

The Audit Report issued by Beosin in no way provides investment advice on any project, nor should it be utilized as investment suggestions of any type. This report represents an extensive evaluation process designed to help our customers improve code quality while mitigating the high risks in Blockchain.

3.4 About BEOSIN

Affiliated to BEOSIN Technology Pte. Ltd., BEOSIN is the first institution in the world specializing in the construction of blockchain security ecosystem. The core team members are all professors, postdocs, PhDs, and Internet elites from world-renowned academic institutions.BEOSIN has more than 20 years of research in formal verification technology, trusted computing, mobile security and kernel security, with overseas experience in studying and collaborating in project research at well-known universities. Through the security audit and defense deployment of more than 2,000 smart contracts, over 50 public blockchains and wallets, and nearly 100 exchanges worldwide, BEOSIN has accumulated rich experience in security attack and defense of the blockchain field, and has developed several security products specifically for blockchain.

BEOSIN
Blockchain Security

Official Website

https://www.beosin.com

Telegram

https://t.me/+dD8Bnqd133RmNWNl

Twitter

https://twitter.com/Beosin_com

Email

Contact@beosin.com