BLOCKSEC

Security Audit Report for Burrowland, Ref-Dcl, Ref-Exchange

Date: Apr 2, 2025 **Version:** 1.2

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Ref Finance
Target	Burrowland, Ref-Dcl, Ref-Exchange

Version History

Version	Date	Description
1.0	May 22, 2024	First release
1.1	Jan 2, 2025	Second release
1.2	Apr 2, 2025	Third release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-04-02 13:09:05+0000

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

Chapter 1 Introduction

1.1 About Target Contracts

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

The target of this audit is the code repository of Burrowland[1], Ref-Dcl[2], Ref-Exchange[3] of Ref Finance. Note that, we did **NOT** audit all the modules in the repository. Specifically, the files covered in this audit include:

```
burrowland/contracts/contract/src/upgrade.rs
burrowland/contracts/contract/src/events.rs
burrowland/contracts/contract/src/price_receiver.rs
burrowland/contracts/contract/src/legacy.rs
burrowland/contracts/contract/src/config.rs
burrowland/contracts/contract/src/lib.rs
burrowland/contracts/contract/src/account_view.rs
burrowland/contracts/contract/src/margin_actions.rs
burrowland/contracts/contract/src/margin_trading.rs
burrowland/contracts/contract/src/margin_pyth.rs
burrowland/contracts/contract/src/margin_config.rs
burrowland/contracts/contract/src/fungible_token.rs
burrowland/contracts/contract/src/big_decimal.rs
burrowland/contracts/contract/src/margin_accounts.rs
burrowland/contracts/contract/src/asset_config.rs
burrowland/contracts/contract/src/account.rs
burrowland/contracts/contract/src/asset_view.rs
burrowland/contracts/contract/src/pyth.rs
burrowland/contracts/contract/src/prices.rs
burrowland/contracts/contract/src/storage.rs
burrowland/contracts/contract/src/shadow_actions.rs
burrowland/contracts/contract/src/margin_position.rs
burrowland/contracts/contract/src/utils.rs
burrowland/contracts/contract/src/margin_base_token_limit.rs
burrowland/contracts/contract/src/protocol_debts.rs
burrowland/contracts/contract/src/storage_keys.rs

ref-contracts/ref-exchange/src/account_deposit.rs
ref-contracts/ref-exchange/src/token_receiver.rs

ref-dcl-lending/contracts/dcl/src/user_asset.rs
ref-dcl-lending/contracts/dcl/src/dcl/utils.rs
ref-dcl-lending/contracts/dcl/src/api/token_receiver.rs
```

[1] https://github.com/burrowHQ/burrowland/tree/margin_trading

[2] https://github.com/ref-finance/ref-dcl/tree/margin_trading

[3] https://github.com/ref-finance/ref-contracts/tree/margin_trading

Listing 1.1: Audit Scope for this Report

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

Project	Version	Commit Hash
Burrowland	Version 1	74462d7e2a299acc0b9702ca278926614c4f4cc8
	Version 2	bfa0b8b75d2d5d978729b22411d44cbdee2156eb
	Version 3	943ea56c7b47856757d57fc18face46d64d8f192
	Version 4	aaaf26979e94617027e9ba72a5c590a498778408
Ref Exchange	Version 1	9c3797aecf58f0f210ebe73b28b9552345e431f7
Ref Dcl	Version 1	70fbc5b70685afc52113636e5154e5dbfd414b65

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.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.

- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

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

1.3.2 DeFi Security

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

1.3.3 NFT Security

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

1.3.4 Additional Recommendation

∗ Gas optimization
∗ Code quality and style

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

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [4] and Common Weakness Enumeration [5]. 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.

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

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

Chapter 2 Findings

In total, we found **ten** potential security issues. Besides, we have **three** recommendations.

- High Risk: 4
- Medium Risk: 1
- Low Risk: 5
- Recommendation: 3

ID	Severity	Description	Category	Status
1	Low	Failure to remove position timestamp from position_latest_actions	Software Security	Fixed
2	High	Lack of sender account check when handling SwapReference message	DeFi Security	Fixed
3	High	Lack of lock when decreasing collateral	DeFi Security	Fixed
4	High	Incorrect enumeration of tokens requiring price feeds	DeFi Security	Fixed
5	Low	Unreasonable pos_id design	DeFi Security	Fixed
6	Low	Lack of reasonable configuration check	DeFi Security	Fixed
7	High	Potential panic during handling message SwapReference	DeFi Security	Fixed
8	Low	Unreasonable check of reserves	DeFi Security	Fixed
9	Low	Potential sandwich attack in force close position token swap	DeFi Security	Confirmed
10	Medium	Unreasonable leverage rate computation	DeFi Security	Fixed
11	-	Automatically construct swap indication from the token information	Recommendation	Confirmed
12	-	Use UnorderedMap for margin_positions instead of HashMap	Recommendation	Fixed
13	-	Incorrect error message in get_token_out()	Recommendation	Fixed

The details are provided in the following sections.
software security

2.1 Software Security

2.1.1 Failure to remove position timestamp from position_latest_actions

Severity Low

Status Fixed in Version 4

Introduced by Version 3

Description The MarginAccount::position_latest_actions variable records the timestamp when a position initiates a swap action. This timestamp is meant to be removed once the swap completes, as it is not utilized in storage fee calculations. However, in the callback_dex_trade() function, which handles failed swap operations, this timestamp is not deleted. As a result, outdated timestamps remain in the `position_latest_actions` map, leading to unnecessary storage consumption.

```
#[private]
pub fn callback_dex_trade(
    &mut self,
    account_id: AccountId,
    pos_id: PosId,
    amount_in: U128,
    pre_token_p_amount: U128,
    op: String,
) {
    let amount_in_used = if let Some(cross_call_result) = promise_result_as_success() {
        serde_json::from_slice::<U128>(&cross_call_result)
            .unwrap()
            .0
    } else {
        0_u128
    };
    if amount_in_used == 0 {
        // trading failed, revert margin operation
        let mut account = self.internal_unwrap_margin_account(&account_id);
        if op == "open" {
            let mt = account.margin_positions.get(&pos_id).unwrap().clone();
            let mut asset_d = self.internal_unwrap_asset(&mt.token_d_id);
            asset_d.margin_pending_debt -= amount_in.0;
            self.internal_set_asset(&mt.token_d_id, asset_d);
            account.deposit_supply_shares(&mt.token_c_id, &mt.token_c_shares);
            // Remove margin_position storage
            account.storage_tracker.start();
            account.margin_positions.remove(&pos_id);
            account.storage_tracker.stop();
            events::emit::margin_open_failed(&account_id, &pos_id);

        } else if op == "decrease" {
            let mut mt = account.margin_positions.get(&pos_id).unwrap();
            let mut asset_p = self.internal_unwrap_asset(&mt.token_p_id);
            let amount_in: Balance = amount_in.into();
            let pre_token_p_amount: Balance = pre_token_p_amount.into();
            if amount_in > pre_token_p_amount {
                asset_p.margin_position += pre_token_p_amount;
                // re-deposit those gap to supply as margin
                let gap = amount_in - pre_token_p_amount;
                let gap_shares = asset_p.supplied.amount_to_shares(gap, false);
                asset_p.supplied.deposit(gap_shares, gap);
                mt.token_c_shares.0 += gap_shares.0;
            } else {
                asset_p.margin_position += amount_in;
            }
            self.internal_set_asset(&mt.token_p_id, asset_p);
            mt.is_locking = false;
            mt.token_p_amount = pre_token_p_amount;
            // Update existing margin_position storage
            account.margin_positions.insert(&pos_id, &mt);
            events::emit::margin_decrease_failed(&account_id, &pos_id);
        }
        self.internal_set_margin_account(&account_id, account);
    }
}
```

Listing 2.1: burrowland/contracts/contract/src/margin_position.rs

Impact Failure to remove timestamps from `position_latest_actions` results in outdated data persisting unnecessarily, leading to redundant storage usage.

Suggestion I Modify the function `callback_dex_trade()` function to ensure the timestamp is properly removed when a swap operation fails.

2.2 DeFi Security

2.2.1 Lack of sender account check when handling SwapReference message

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `ft_on_transfer()` does not check if the `sender` is a registered `Ref V1` or `Ref V2` contract when handling the `SwapReference` message.

```
TokenReceiverMsg::SwapReference { swap_ref } => {
    let mut account = self.internal_unwrap_margin_account(&swap_ref.account_id);
    if swap_ref.op == "open" {
        self.on_open_trade_return(&mut account, amount, &swap_ref);
    } else if swap_ref.op == "decrease"
        || swap_ref.op == "close"
        || swap_ref.op == "liquidate"
        || swap_ref.op == "forceclose"
    {
        let event = self.on_decrease_trade_return(&mut account, amount, &swap_ref);
        events::emit::margin_decrease_succeeded(&swap_ref.op, event);
    }
    self.internal_set_margin_account(&swap_ref.account_id, account);
    return PromiseOrValue::Value(U128(0));
}
```

Listing 2.2: burrowland/contracts/contract/src/fungible_token.rs

Impact An attacker can impersonate a `REF V1` or `REF V2` contract by simply sending the custom `SwapReference` message to the contract. The contract would then correspondingly reduce the user's debt and add assets, while the attacker does not incur any cost.

Suggestion I Add a check to ensure that the `sender` must be `REF v1` or `REF V2` contract.

2.2.2 Lack of lock when decreasing collateral

Severity High

Status Fixed in Version 2

Introduced by Version 1

Description When a user opens a margin position, the token_c_shares value is set appropriately, while the token_d_shares and token_p_amount values are set to zero. These token_d_shares and token_p_amount values are intended to be reset upon receiving the SwapReference message from the Ref V1 or Ref V2 contract. In between these two processes, this margin position is locked.

However, the function internal_margin_decrease_collateral() does not check if the margin position is locked. This allows the user to remove almost all of the collateral, putting the margin position in a state where it can be liquidated or force closed.

```rust
pub(crate) fn internal_margin_decrease_collateral(
    &mut self,
    account: &mut MarginAccount,
    pos_id: &PosId,
    amount: Balance,
    prices: &Prices,
) -> AccountId {
    let margin_config = self.internal_margin_config();
    let mut mt = account
        .margin_positions
        .get(pos_id)
        .expect("Position not exist")
        .clone();
    let token_id = mt.token_c_id.clone();
    let asset = self.internal_unwrap_asset(&mt.token_c_id);
    let shares = asset.supplied.amount_to_shares(amount, true);

    // collateral can NOT decrease to 0
    assert!(
        mt.token_c_shares.0 > shares.0,
        "Not enough collateral to decrease"
    );
    mt.token_c_shares.0 -= shares.0;

    assert!(
        !self.is_mt_liquidatable(&mt, prices, margin_config.min_safty_buffer),
        "Margin position would be below liquidation line"
    );
    assert!(
        !self.is_mt_forcecloseable(&mt, prices),
        "Margin position would be below forceclose line"
    );

    assert!(
        self.get_mtp_lr(&mt, prices).unwrap()
            <= BigDecimal::from(margin_config.max_leverage_rate as u32),
        "Leverage rate is too high"
    );

    account.deposit_supply_shares(&mt.token_c_id, &shares);
    account.margin_positions.insert(pos_id.clone(), mt);

    token_id
}
```

Listing 2.3: burrowland/contracts/contract/src/margin_actions.rs

Impact After opening the position, an attacker can immediately withdraw almost all of the collateral and trigger the swap action of the contract. Despite having slippage protection, there is still some room for a sandwich attack during this swap.

When `Ref V1` or `Ref V2` returns the swap result, the margin position may be in force-closable state and need to be closed using the contract's `reserves`, resulting in a loss of the contract's assets.

Suggestion I Add a check to ensure that the margin position is not locked before operating in the function `internal_margin_decrease_collateral`.

2.2.3 Incorrect enumeration of tokens requiring price feeds

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `margin_involved_tokens()` is intended to enumerate the tokens that require price feeds for a given `MarginAction`s. It adds the `token_c_id` and `token_d_id` of the relevant margin positions. However, according to the design, the `token_c_id` may be the same as the `token_d_id`. In this case, the `token_p_id`, whose price is needed for calculating slippage protection, is not included in this enumeration.

```
pub fn margin_involved_tokens(&self, account: &MarginAccount, actions: &Vec<MarginAction>) ->
      Vec<TokenId> {
    let mut tokens = HashSet::new();
    actions.iter().for_each(|action|{
        let pos_id = match action {
            MarginAction::DecreaseCollateral { pos_id, amount: _ } => {
                Some(pos_id)
            }
            MarginAction::OpenPosition { token_c_id, token_c_amount: _, token_d_id,
                 token_d_amount: _, token_p_id: _, min_token_p_amount: _, swap_indication: _ }
                 => {
                tokens.insert(token_c_id.clone());
                tokens.insert(token_d_id.clone());
                None
            },
            MarginAction::DecreaseMTPosition { pos_id, token_p_amount: _, min_token_d_amount: _
                 , swap_indication: _ }=> {
                Some(pos_id)
            }
            MarginAction::CloseMTPosition { pos_id, token_p_amount: _, min_token_d_amount: _, swap_indication: _ } => {
                Some(pos_id)
            }
            MarginAction::LiquidateMTPosition { pos_owner_id, pos_id, token_p_amount: _, min_token_d_amount: _, swap_indication: _ } => {
                let pos_owner_account = self.internal_get_margin_account(pos_owner_id).expect("Margin account not exist");
                let mt = pos_owner_account.margin_positions.get(pos_id).expect("Position not exist");
                tokens.insert(mt.token_c_id.clone());
                tokens.insert(mt.token_d_id.clone());
                None
            }
            MarginAction::ForceCloseMTPosition { pos_owner_id, pos_id, token_p_amount: _, min_token_d_amount: _, swap_indication: _ } => {
                let pos_owner_account = self.internal_get_margin_account(pos_owner_id).expect("Margin account not exist");
                let mt = pos_owner_account.margin_positions.get(pos_id).expect("Position not exist");
                tokens.insert(mt.token_c_id.clone());
                tokens.insert(mt.token_d_id.clone());
                None
            }
            _ => None
        };
        if let Some(pos_id) = pos_id {
            let mt = account.margin_positions.get(pos_id).expect("Position not exist");
            tokens.insert(mt.token_c_id.clone());
            tokens.insert(mt.token_d_id.clone());
        }
    });
    tokens.into_iter().collect()
}
```

Listing 2.4: burrowland/contracts/contract/src/margin_pyth.rs

Impact The callback function `callback_margin_execute_with_pyth()` that is invoked after `internal_margin_execute_with_pyth()` may fail due to missing price feeds for certain tokens required by the margin action.

Suggestion I Modify the function `margin_involved_tokens()` to replace `token_c_id` with `token_p_id`, so that it adds the position token (`token_p_id`) and debt token (`token_d_id`) to the list of tokens requiring price feeds.

2.2.4 Unreasonable pos_id design

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The pos_id is generated by concatenating the account_id with the env::block_timestamp(). This means that within the same block, all generated pos_ids with the same margin account will be identical.

```rust
pub(crate) fn internal_margin_open_position(
    &mut self,
    ts: Timestamp,
    account: &mut MarginAccount,
    token_c_id: &AccountId,
    token_c_amount: Balance,
    token_d_id: &AccountId,
    token_d_amount: Balance,
    token_p_id: &AccountId,
    min_token_p_amount: Balance,
    swap_indication: &SwapIndication,
    prices: &Prices,
) -> EventDataMarginOpen {
    let pos_id = format!("{}_{}", account.account_id.clone(), ts);
    assert!(
        !account.margin_positions.contains_key(&pos_id),
        "Margin position already exist"
    );

    let asset_c = self.internal_unwrap_asset(token_c_id);
    let asset_p = self.internal_unwrap_asset(token_p_id);
    let mut asset_d = self.internal_unwrap_asset(token_d_id);
    let margin_config = self.internal_margin_config();

    // check legitimacy: assets legal; swap_indication matches;
    margin_config.check_pair(&token_d_id, &token_p_id, &token_c_id);
    let mut swap_detail = self.parse_swap_indication(swap_indication);
    let ft_d_amount = token_d_amount / 10u128.pow(asset_d.config.extra_decimals as u32);
    assert!(
        swap_detail.verify_token_in(token_d_id, ft_d_amount),
        "token_in check failed"
    );
    let ft_p_amount =
        min_token_p_amount / 10u128.pow(asset_p.config.extra_decimals as u32);
    assert!(
        swap_detail.verify_token_out(token_p_id, ft_p_amount),
        "token_out check failed"
    );

    // check safty:
    //  min_position_amount reasonable
    assert!(
        is_min_amount_out_reasonable(
            token_d_amount,
            &asset_d,
            prices.get_unwrap(&token_d_id),
            &asset_p,
            prices.get_unwrap(&token_p_id),
            min_token_p_amount,
            margin_config.max_slippage_rate,
        ),
        "min_position_amount is too low"
    );
    //  margin_hf more than 1 + safty_buffer_rate(10%)
    let mut mt = MarginTradingPosition::new(
        ts,
        token_c_id.clone(),
        asset_c.supplied.amount_to_shares(token_c_amount, false),
        token_d_id.clone(),
        token_p_id.clone(),
    );
    mt.token_d_shares = asset_d.margin_debt.amount_to_shares(token_d_amount, true);
    mt.token_p_amount = min_token_p_amount;
    assert!(
        !self.is_mt_liquidatable(&mt, prices, margin_config.min_safty_buffer),
        "Debt is too much"
    );
    assert!(
        !self.is_mt_forcecloseable(&mt, prices),
        "Debt is too much"
    );
    //  leverage rate less than max leverage rate
    assert!(
        self.get_mtp_lr(&mt, prices).unwrap()
            <= BigDecimal::from(margin_config.max_leverage_rate as u32),
        "Leverage rate is too high"
    );

    // passes all check, start to open
    let event = EventDataMarginOpen {
        account_id: account.account_id.clone(),
        pos_id: pos_id.clone(),
        token_c_id: token_c_id.clone(),
        token_c_amount,
        token_c_shares: mt.token_c_shares,
        token_d_id: token_d_id.clone(),
        token_d_amount,
        token_p_id: token_p_id.clone(),
        token_p_amount: min_token_p_amount,
    };
    account.withdraw_supply_shares(token_c_id, &mt.token_c_shares);
    mt.token_d_shares.0 = 0;
    mt.token_p_amount = 0;
    asset_d.increase_margin_pending_debt(token_d_amount, margin_config.pending_debt_scale);
    self.internal_set_asset(token_d_id, asset_d);
    // TODO: may need to change to store in an unorderedmap in user Account
    account.margin_positions.insert(pos_id.clone(), mt);

    // step 4: call dex to trade and wait for callback
    // organize swap action
    let swap_ref = SwapReference {
        account_id: account.account_id.clone(),
        pos_id: pos_id.clone(),
        amount_in: token_d_amount.into(),
        op: format!("open"),
        liquidator_id: None,
    };
    swap_detail.set_client_echo(&swap_ref.to_msg_string());
    let swap_msg = swap_detail.to_msg_string();
    ext_fungible_token::ext(token_d_id.clone())
        .with_attached_deposit(1)
        .with_static_gas(GAS_FOR_FT_TRANSFER_CALL)
        .ft_transfer_call(
            swap_indication.dex_id.clone(),
            U128(ft_d_amount),
            None,
            swap_msg,
        )
        .then(
            Self::ext(env::current_account_id())
                .with_static_gas(GAS_FOR_FT_TRANSFER_CALL_CALLBACK)
                .callback_dex_trade(
                    account.account_id.clone(),
                    pos_id.clone(),
                    token_d_amount.into(),
                    U128(0),
                    format!("open"),
                ),
        );
        event
}
```

Listing 2.5: burrowland/contracts/contract/src/margin_position.rs

Impact Users are unable to open multiple margin positions within the same block, as all the `pos_id`s generated in that block will be identical.

Suggestion I Use a more reasonable method to generate the `pos_id`, such as an auto-incrementing variable.

2.2.5 Lack of reasonable configuration check

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The newly introduced `MarginConfig` contains many global parameters for managing margin trading. These global parameters are used to compare against the user-provided parameters, ensuring that the user does not input values that are too large or too small. However, some of these parameters lack the necessary range checks. As shown in the table below:

In addition, the variable "safty" is a typo of "safety".

```
#[payable]
pub fn update_max_leverage_rate(&mut self, max_leverage_rate: u8) {
```

Parameter	Reasonable Range
max_leverage_rate	bigger than 1
pending_debt_scale	(0, MAX_RATIO)
max_slippage_rate	(0, MAX_RATIO)
min_safty_buffer	(0, MAX_RATIO)
margin_debt_discount_rate	(0, MAX_RATIO)
open_position_fee_rate	(0, MAX_RATIO)

```
        assert_one_yocto();
        self.assert_owner();
        let mut mc = self.internal_margin_config();
        mc.max_leverage_rate = max_leverage_rate;
        self.margin_config.set(&mc);
    }

    #[payable]
    pub fn update_pending_debt_scale(&mut self, pending_debt_scale: u32) {
        assert_one_yocto();
        self.assert_owner();
        let mut mc = self.internal_margin_config();
        mc.pending_debt_scale = pending_debt_scale;
        self.margin_config.set(&mc);
    }

    #[payable]
    pub fn update_max_slippage_rate(&mut self, max_slippage_rate: u32) {
        assert_one_yocto();
        self.assert_owner();
        let mut mc = self.internal_margin_config();
        mc.max_slippage_rate = max_slippage_rate;
        self.margin_config.set(&mc);
    }

    #[payable]
    pub fn update_min_safty_buffer(&mut self, min_safty_buffer: u32) {
        assert_one_yocto();
        self.assert_owner();
        let mut mc = self.internal_margin_config();
        mc.min_safty_buffer = min_safty_buffer;
        self.margin_config.set(&mc);
    }

    #[payable]
    pub fn update_margin_debt_discount_rate(&mut self, margin_debt_discount_rate: u32) {
        assert_one_yocto();
        self.assert_owner();
        let mut mc = self.internal_margin_config();
        mc.margin_debt_discount_rate = margin_debt_discount_rate;
        self.margin_config.set(&mc);
    }

    #[payable]
```

```
pub fn update_open_position_fee_rate(&mut self, open_position_fee_rate: u32) {
    assert_one_yocto();
    self.assert_owner();
    let mut mc = self.internal_margin_config();
    mc.open_position_fee_rate = open_position_fee_rate;
    self.margin_config.set(&mc);
}
```

Listing 2.6: burrowland/contracts/contract/src/margin_config.rs

Impact Unreasonable configuration values due to missing range checks can cause the contract to not work as intended.

Suggestion I Add corresponding checks according to the table.

2.2.6 Potential panic during handling message SwapReference

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description When handling the `SwapReference` message, the transaction should not fail. Otherwise, the user would lose the assets they obtained from the `Ref V1` or `Ref V2` contract, causing an inconsistency in the contract state. The corresponding margin position would also remain locked and unable to be unlocked. However, during this process, the function `internal_set_margin_account()` needs to be invoked to write back the account states of the `liquidator` and `account`, which includes storage checks that may cause a panic by insufficient storage fees.

```
TokenReceiverMsg::SwapReference { swap_ref } => {
    let mut account = self.internal_unwrap_margin_account(&swap_ref.account_id);
    if swap_ref.op == "open" {
        self.on_open_trade_return(&mut account, amount, &swap_ref);
    } else if swap_ref.op == "decrease"
        || swap_ref.op == "close"
        || swap_ref.op == "liquidate"
        || swap_ref.op == "forceclose"
    {
        let event = self.on_decrease_trade_return(&mut account, amount, &swap_ref);
        events::emit::margin_decrease_succeeded(&swap_ref.op, event);
    }
    self.internal_set_margin_account(&swap_ref.account_id, account);
    return PromiseOrValue::Value(U128(0));
}
```

Listing 2.7: burrowland/contracts/contract/src/fungible_token.rs

```
pub(crate) fn on_decrease_trade_return(
    &mut self,
    account: &mut MarginAccount,
    amount: Balance,
    sr: &SwapReference,
    ) -> EventDataMarginDecreaseResult {
        let mut mt = account.margin_positions.get(&sr.pos_id).unwrap().clone();
        let mut asset_debt = self.internal_unwrap_asset(&mt.token_d_id);
        let mut asset_position = self.internal_unwrap_asset(&mt.token_p_id);
        let (mut benefit_m_shares, mut benefit_d_shares, mut benefit_p_shares) =
            (0_u128, 0_u128, 0_u128);

        // figure out actual repay amount and shares
        // figure out how many debt_cap been repaid, and charge corresponding holding-position fee from repayment.
        let debt_amount = asset_debt
            .margin_debt
            .shares_to_amount(mt.token_d_shares, true);
        let hp_fee = u128_ratio(
            mt.debt_cap,
            asset_debt.unit_acc_hp_interest - mt.uahpi_at_open,
            UNIT,
        );
        let repay_cap = u128_ratio(mt.debt_cap, amount, debt_amount + hp_fee);

        let (repay_amount, repay_shares, left_amount, repay_hp_fee) = if repay_cap >= mt.debt_cap {
            (debt_amount, mt.token_d_shares, amount - debt_amount - hp_fee, hp_fee)
        } else {
            let repay_hp_fee = u128_ratio(hp_fee, repay_cap, mt.debt_cap);
            (
                amount - repay_hp_fee,
                asset_debt
                    .margin_debt
                    .amount_to_shares(amount - repay_hp_fee, false),
                0,
                repay_hp_fee,
            )
        };
        asset_debt.margin_debt.withdraw(repay_shares, repay_amount);
        mt.token_d_shares.0 -= repay_shares.0;
        mt.debt_cap = if repay_cap >= mt.debt_cap {
            0
        } else {
            mt.debt_cap - repay_cap
        };
        // distribute hp_fee
        asset_debt.prot_fee += repay_hp_fee;

        // handle possible leftover debt asset, put them into user's supply
        if left_amount > 0 {
            let supply_shares = asset_debt.supplied.amount_to_shares(left_amount, false);
            if supply_shares.0 > 0 {
                asset_debt.supplied.deposit(supply_shares, left_amount);
                benefit_d_shares = supply_shares.0;
            }
        }

        if sr.op != "decrease" {
            // try to repay remaining debt from margin
            if mt.token_d_shares.0 > 0 && mt.token_d_id == mt.token_c_id {
                let remain_debt_balance = asset_debt
                    .margin_debt
                    .shares_to_amount(mt.token_d_shares, true);
                let margin_shares_to_repay = asset_debt
                    .supplied
                    .amount_to_shares(remain_debt_balance, true);
                let (repay_debt_share, used_supply_share, repay_amount) =
                    if margin_shares_to_repay <= mt.token_c_shares {
                        (mt.token_d_shares, margin_shares_to_repay, remain_debt_balance)
                    } else {
                        // use all margin balance to repay
                        let margin_balance = asset_debt
                            .supplied
                            .shares_to_amount(mt.token_c_shares, false);
                        let repay_debt_shares = asset_debt
                            .margin_debt
                            .amount_to_shares(margin_balance, false);
                        (repay_debt_shares, mt.token_c_shares, margin_balance)
                    };
                asset_debt
                    .supplied
                    .withdraw(used_supply_share, repay_amount);
                asset_debt
                    .margin_debt
                    .withdraw(repay_debt_share, repay_amount);
                mt.token_d_shares.0 -= repay_debt_share.0;
                mt.token_c_shares.0 -= used_supply_share.0;
            }
        }

        if sr.op == "forceclose" {
            // try to use protocol reserve to repay remaining debt
            if mt.token_d_shares.0 > 0 {
                let remain_debt_balance = asset_debt
                    .margin_debt
                    .shares_to_amount(mt.token_d_shares, true);
                if asset_debt.reserved > remain_debt_balance {
                    asset_debt.reserved -= remain_debt_balance;
                    asset_debt
                        .margin_debt
                        .withdraw(mt.token_d_shares, remain_debt_balance);
                    mt.token_d_shares.0 = 0;
                }
            }
        }

        mt.is_locking = false;
        account
            .margin_positions
            .insert(sr.pos_id.clone(), mt.clone());

        let event = EventDataMarginDecreaseResult {
            account_id: account.account_id.clone(),
            pos_id: sr.pos_id.clone(),
            liquidator_id: sr.liquidator_id.clone(),
            token_c_id: mt.token_c_id.clone(),
            token_c_shares: mt.token_c_shares,
            token_d_id: mt.token_d_id.clone(),
            token_d_shares: mt.token_d_shares,
            token_p_id: mt.token_p_id.clone(),
            token_p_amount: mt.token_p_amount,
            holding_fee: repay_hp_fee,
        };

        // try to settle this position
        if mt.token_d_shares.0 == 0 {
            // close this position and remaining asset goes back to user's inner account
            // TODO: change to directly send assets back to user
            if mt.token_c_shares.0 > 0 {
                benefit_m_shares = mt.token_c_shares.0;
            }
            if mt.token_p_amount > 0 {
                let position_shares = asset_position
                    .supplied
                    .amount_to_shares(mt.token_p_amount, false);
                asset_position
                    .supplied
                    .deposit(position_shares, mt.token_p_amount);
                asset_position.margin_position -= mt.token_p_amount;
                benefit_p_shares = position_shares.0;
            }
            account.margin_positions.remove(&sr.pos_id);
        } else {
            if sr.op != "decrease" {
                env::log_str(&format!(
                    "{} failed due to insufficient fund, user {}, pos_id {}",
                    sr.op.clone(),
                    account.account_id.clone(),
                    sr.pos_id.clone()
                ));
            }
        }

        // distribute benefits
        if benefit_d_shares > 0 || benefit_m_shares > 0 || benefit_p_shares > 0 {
            if sr.op == "liquidate" || sr.op == "forceclose" {
                let mut liquidator_account =
                    if let Some(ref liquidator_account_id) = sr.liquidator_id {
                        if let Some(x) = self.internal_get_margin_account(&liquidator_account_id) {
                            x
                        } else {
                            self.internal_unwrap_margin_account(&self.internal_config().owner_id)
                        }
                    } else {
                        self.internal_unwrap_margin_account(&self.internal_config().owner_id)
                    };
                if benefit_d_shares > 0 {
                    liquidator_account
                        .deposit_supply_shares(&mt.token_d_id, &U128(benefit_d_shares));
                }
                if benefit_m_shares > 0 {
                    liquidator_account
                        .deposit_supply_shares(&mt.token_c_id, &U128(benefit_m_shares));
                }
                if benefit_p_shares > 0 {
                    liquidator_account
                        .deposit_supply_shares(&mt.token_p_id, &U128(benefit_p_shares));
                }
                self.internal_set_margin_account(
                    &liquidator_account.account_id.clone(),
                    liquidator_account,
                );
            } else {
                if benefit_d_shares > 0 {
                    account.deposit_supply_shares(&mt.token_d_id, &U128(benefit_d_shares));
                }
                if benefit_m_shares > 0 {
                    account.deposit_supply_shares(&mt.token_c_id, &U128(benefit_m_shares));
                }
                if benefit_p_shares > 0 {
                    account.deposit_supply_shares(&mt.token_p_id, &U128(benefit_p_shares));
                }
            }
        }

        self.internal_set_asset(&mt.token_d_id, asset_debt);
        self.internal_set_asset(&mt.token_p_id, asset_position);

        event
    }
```

Listing 2.8: burrowland/contracts/contract/src/margin_trading.rs

Impact Users may lose the assets they obtained from the `Ref V1` or `Ref V2` contract, causing an inconsistency in the contract state. The corresponding margin position would also remain locked and unable to be unlocked.

Suggestion I Revise the corresponding logic.

2.2.7 Unreasonable check of reserves

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description When executing a force close operation in the function `on_decrease_trade_return()`, any remaining `debt` needs to be covered by the contract's `reserves`. However, when comparing the amount between the contract's `reserves` and the remaining `debt` amount, the check requires the `reserves` to be strictly greater than the remaining `debt`.

```rust
pub(crate) fn on_decrease_trade_return(
    &mut self,
    account: &mut MarginAccount,
    amount: Balance,
    sr: &SwapReference,
) -> EventDataMarginDecreaseResult {
    let mut mt = account.margin_positions.get(&sr.pos_id).unwrap().clone();
    let mut asset_debt = self.internal_unwrap_asset(&mt.token_d_id);
    let mut asset_position = self.internal_unwrap_asset(&mt.token_p_id);
    let (mut benefit_m_shares, mut benefit_d_shares, mut benefit_p_shares) =
        (0_u128, 0_u128, 0_u128);

    // figure out actual repay amount and shares
    // figure out how many debt_cap been repaid, and charge corresponding holding-position fee from repayment.
    let debt_amount = asset_debt
        .margin_debt
        .shares_to_amount(mt.token_d_shares, true);
    let hp_fee = u128_ratio(
        mt.debt_cap,
        asset_debt.unit_acc_hp_interest - mt.uahpi_at_open,
        UNIT,
    );
    let repay_cap = u128_ratio(mt.debt_cap, amount, debt_amount + hp_fee);

    let (repay_amount, repay_shares, left_amount, repay_hp_fee) = if repay_cap >= mt.debt_cap {
        (debt_amount, mt.token_d_shares, amount - debt_amount - hp_fee, hp_fee)
    } else {
        let repay_hp_fee = u128_ratio(hp_fee, repay_cap, mt.debt_cap);
        (
            amount - repay_hp_fee,
            asset_debt
                .margin_debt
                .amount_to_shares(amount - repay_hp_fee, false),
            0,
            repay_hp_fee,
        )
    };
    asset_debt.margin_debt.withdraw(repay_shares, repay_amount);
    mt.token_d_shares.0 -= repay_shares.0;
    mt.debt_cap = if repay_cap >= mt.debt_cap {
        0
    } else {
        mt.debt_cap - repay_cap
    };
    // distribute hp_fee
```

```
        asset_debt.prot_fee += repay_hp_fee;

        // handle possible leftover debt asset, put them into user's supply
        if left_amount > 0 {
            let supply_shares = asset_debt.supplied.amount_to_shares(left_amount, false);
            if supply_shares.0 > 0 {
                asset_debt.supplied.deposit(supply_shares, left_amount);
                benefit_d_shares = supply_shares.0;
            }
        }

        if sr.op != "decrease" {
            // try to repay remaining debt from margin
            if mt.token_d_shares.0 > 0 && mt.token_d_id == mt.token_c_id {
                let remain_debt_balance = asset_debt
                    .margin_debt
                    .shares_to_amount(mt.token_d_shares, true);
                let margin_shares_to_repay = asset_debt
                    .supplied
                    .amount_to_shares(remain_debt_balance, true);
                let (repay_debt_share, used_supply_share, repay_amount) =
                    if margin_shares_to_repay <= mt.token_c_shares {
                        (mt.token_d_shares, margin_shares_to_repay, remain_debt_balance)
                    } else {
                        // use all margin balance to repay
                        let margin_balance = asset_debt
                            .supplied
                            .shares_to_amount(mt.token_c_shares, false);
                        let repay_debt_shares = asset_debt
                            .margin_debt
                            .amount_to_shares(margin_balance, false);
                        (repay_debt_shares, mt.token_c_shares, margin_balance)
                    };
                asset_debt
                    .supplied
                    .withdraw(used_supply_share, repay_amount);
                asset_debt
                    .margin_debt
                    .withdraw(repay_debt_share, repay_amount);
                mt.token_d_shares.0 -= repay_debt_share.0;
                mt.token_c_shares.0 -= used_supply_share.0;
            }
        }

        if sr.op == "forceclose" {
            // try to use protocol reserve to repay remaining debt
            if mt.token_d_shares.0 > 0 {
                let remain_debt_balance = asset_debt
                    .margin_debt
                    .shares_to_amount(mt.token_d_shares, true);
                if asset_debt.reserved > remain_debt_balance {
                    asset_debt.reserved -= remain_debt_balance;
                    asset_debt
```

```
                        .margin_debt
                        .withdraw(mt.token_d_shares, remain_debt_balance);
                    mt.token_d_shares.0 = 0;
                }
            }
        }

        mt.is_locking = false;
        account
            .margin_positions
            .insert(sr.pos_id.clone(), mt.clone());


        let event = EventDataMarginDecreaseResult {
            account_id: account.account_id.clone(),
            pos_id: sr.pos_id.clone(),
            liquidator_id: sr.liquidator_id.clone(),
            token_c_id: mt.token_c_id.clone(),
            token_c_shares: mt.token_c_shares,
            token_d_id: mt.token_d_id.clone(),
            token_d_shares: mt.token_d_shares,
            token_p_id: mt.token_p_id.clone(),
            token_p_amount: mt.token_p_amount,
            holding_fee: repay_hp_fee,
        };

        // try to settle this position
        if mt.token_d_shares.0 == 0 {
            // close this position and remaining asset goes back to user's inner account
            // TODO: change to directly send assets back to user
            if mt.token_c_shares.0 > 0 {
                benefit_m_shares = mt.token_c_shares.0;
            }
            if mt.token_p_amount > 0 {
                let position_shares = asset_position
                    .supplied
                    .amount_to_shares(mt.token_p_amount, false);
                asset_position
                    .supplied
                    .deposit(position_shares, mt.token_p_amount);
                asset_position.margin_position -= mt.token_p_amount;
                benefit_p_shares = position_shares.0;
            }
            account.margin_positions.remove(&sr.pos_id);
        } else {
            if sr.op != "decrease" {
                env::log_str(&format!(
                    "{} failed due to insufficient fund, user {}, pos_id {}",
                    sr.op.clone(),
                    account.account_id.clone(),
                    sr.pos_id.clone()
                ));
            }
        }

        // distribute benefits
        if benefit_d_shares > 0 || benefit_m_shares > 0 || benefit_p_shares > 0 {
            if sr.op == "liquidate" || sr.op == "forceclose" {
                let mut liquidator_account =
                    if let Some(ref liquidator_account_id) = sr.liquidator_id {
                        if let Some(x) = self.internal_get_margin_account(&liquidator_account_id) {
                            x
                        } else {
                            self.internal_unwrap_margin_account(&self.internal_config().owner_id)
                        }
                    } else {
                        self.internal_unwrap_margin_account(&self.internal_config().owner_id)
                    };
                if benefit_d_shares > 0 {
                    liquidator_account
                        .deposit_supply_shares(&mt.token_d_id, &U128(benefit_d_shares));
                }
                if benefit_m_shares > 0 {
                    liquidator_account
                        .deposit_supply_shares(&mt.token_c_id, &U128(benefit_m_shares));
                }
                if benefit_p_shares > 0 {
                    liquidator_account
                        .deposit_supply_shares(&mt.token_p_id, &U128(benefit_p_shares));
                }
                self.internal_set_margin_account(
                    &liquidator_account.account_id.clone(),
                    liquidator_account,
                );
            } else {
                if benefit_d_shares > 0 {
                    account.deposit_supply_shares(&mt.token_d_id, &U128(benefit_d_shares));
                }
                if benefit_m_shares > 0 {
                    account.deposit_supply_shares(&mt.token_c_id, &U128(benefit_m_shares));
                }
                if benefit_p_shares > 0 {
                    account.deposit_supply_shares(&mt.token_p_id, &U128(benefit_p_shares));
                }
            }
        }

        self.internal_set_asset(&mt.token_d_id, asset_debt);
        self.internal_set_asset(&mt.token_p_id, asset_position);

        event
    }
```

Listing 2.9: burrowland/contracts/contract/src/margin_trading.rs

Impact Margin positions that can be forced closed may fail to be closed.

Suggestion I Revise the check to be greater or equal.

2.2.8 Potential sandwich attack in force close position token swap

Severity Low

Status Confirmed

Introduced by `Version 1`

Description The force closing margin position operation swaps the liquidated user's `position` token into the `debt` token. Despite having slippage protection, there is still some room for a sandwich attack during this swap.

This issue is similar to issue-2, where both involve a potential sandwich attack on a token swap. However, issue-2 can be reliably triggered by an attacker, whereas under normal circumstances, the probability of a margin position reaching a force-closable state to trigger this swap is relatively low.

```rust
pub(crate) fn process_decrease_margin_position(
    &mut self,
    account: &mut MarginAccount,
    pos_id: &String,
    token_p_amount: Balance,
    min_token_d_amount: Balance,
    swap_indication: &SwapIndication,
    prices: &Prices,
    op: String,
    liquidator_id: Option<AccountId>,
) -> EventDataMarginDecrease {
    let mut mt = account
        .margin_positions
        .get_mut(pos_id)
        .expect("Position not exist");
    assert!(
        !mt.is_locking,
        "Position is currently waiting for a trading result."
    );
    let pre_token_p_amount = mt.token_p_amount;
    let mut asset_p = self.internal_unwrap_asset(&mt.token_p_id);
    let asset_d = self.internal_unwrap_asset(&mt.token_d_id);
    let margin_config = self.internal_margin_config();

    //  check swap_indication
    let mut swap_detail = self.parse_swap_indication(swap_indication);
    let ft_p_amount =
        token_p_amount / 10u128.pow(asset_p.config.extra_decimals as u32);
    assert!(
        swap_detail.verify_token_in(&mt.token_p_id, ft_p_amount),
        "token_in check failed"
    );
    let ft_d_amount = min_token_d_amount / 10u128.pow(asset_d.config.extra_decimals as u32);
    assert!(
        swap_detail.verify_token_out(&mt.token_d_id, ft_d_amount),
        "token_out check failed"
    );

    //  min_debt_amount reasonable
    assert!(
        is_min_amount_out_reasonable(
            token_p_amount,
            &asset_p,
            prices.get_unwrap(&mt.token_p_id),
            &asset_d,
            prices.get_unwrap(&mt.token_d_id),
            min_token_d_amount,
            margin_config.max_slippage_rate,
        ),
        "min_debt_amount is too low"
    );

    if op == "close" || op == "liquidate" {
        //  ensure all debt would be repaid
        //  and take holding-position fee into account
        let total_debt_amount = asset_d
            .margin_debt
            .shares_to_amount(mt.token_d_shares, true);
        let hp_fee = u128_ratio(
            mt.debt_cap,
            asset_d.unit_acc_hp_interest - mt.uahpi_at_open,
            UNIT,
        );
        if min_token_d_amount < total_debt_amount + hp_fee {
            assert_eq!(
                mt.token_c_id, mt.token_d_id,
                "Can NOT trade under total debt when margin and debt asset are not the same"
            );
            let gap_shares = asset_d
                .supplied
                .amount_to_shares(total_debt_amount + hp_fee - min_token_d_amount, true);
            assert!(
                mt.token_c_shares.0 > gap_shares.0,
                "Not all debt could be repaid"
            );
        }
    }

    if op == "liquidate" {
        assert!(
            self.is_mt_liquidatable(&mt, prices, margin_config.min_safty_buffer),
            "Margin position is not liquidatable"
        );
    } else if op == "forceclose" {
        assert!(
            self.is_mt_forcecloseable(&mt, prices),
            "Margin position is not forceclose-able"
        );
    }

    //  ensure enough position token to trade
    if token_p_amount > mt.token_p_amount {
        // try to add some of margin asset into trading
        assert_eq!(
            mt.token_c_id, mt.token_p_id,
            "Not enough position asset balance"
        );
        let gap_shares = asset_p
            .supplied
            .amount_to_shares(token_p_amount - mt.token_p_amount, true);
        mt.token_c_shares
            .0
            .checked_sub(gap_shares.0)
            .expect("Not enough position asset balance");
        asset_p
            .supplied
            .withdraw(gap_shares, token_p_amount - mt.token_p_amount);
        asset_p.margin_position -= mt.token_p_amount;
        mt.token_p_amount = 0;
    } else {
        asset_p.margin_position -= token_p_amount;
        mt.token_p_amount -= token_p_amount;
    }

    // prepare to close
    mt.is_locking = true;
    self.internal_set_asset(&mt.token_p_id, asset_p);
    // TODO: mt may be needed to change to store in an unorderedmap in user Account

    let event = EventDataMarginDecrease {
        account_id: account.account_id.clone(),
        pos_id: pos_id.clone(),
        liquidator_id: liquidator_id.clone(),
        token_p_id: mt.token_p_id.clone(),
        token_p_amount,
        token_d_id: mt.token_d_id.clone(),
        token_d_amount: min_token_d_amount,
    };

    // step 3: call dex to trade and wait for callback
    // organize swap action
    let swap_ref = SwapReference {
        account_id: account.account_id.clone(),
        pos_id: pos_id.clone(),
        amount_in: token_p_amount.into(),
        op,
        liquidator_id,
    };
    swap_detail.set_client_echo(&swap_ref.to_msg_string());
    let swap_msg = swap_detail.to_msg_string();
    ext_fungible_token::ext(mt.token_p_id.clone())
        .with_attached_deposit(1)
        .with_static_gas(GAS_FOR_FT_TRANSFER_CALL)
        .ft_transfer_call(
            swap_indication.dex_id.clone(),
            U128(ft_p_amount),
            None,
            swap_msg,
        )
        .then(
            Self::ext(env::current_account_id())
                .with_static_gas(GAS_FOR_FT_TRANSFER_CALL_CALLBACK)
                .callback_dex_trade(
                    account.account_id.clone(),
                    pos_id.clone(),
                    token_p_amount.into(),
                    pre_token_p_amount.into(),
                    format!("decrease"),
                ),
        );
    event
}
```

Listing 2.10: burrowland/contracts/contract/src/margin_position.rs

Impact An attacker can set the `min_amount_d_out` to be as small as possible within the allowed slippage protection range. This enables them to conduct a sandwich attack, forcing the contract to use more of its reserves to cover the liquidation.

Suggestion I The value of the parameter `max_slippage_rate` for slippage protection should be set more strictly (smaller) when it comes to force closing, distinguishing it from the values used for opening or other operations.

Feedback from the Project Accept this situation and try to ensure success when the force closing is initiated.

2.2.9 Unreasonable leverage rate computation

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The hold position fee is not included in leverage rate computation in the function `get_mtp_lr`.

```
pub(crate) fn get_mtp_lr(
    &self,
    mt: &MarginTradingPosition,
    prices: &Prices,
) -> Option<BigDecimal> {
    if mt.token_c_shares.0 == 0 || mt.token_d_shares.0 == 0 {
        None
    } else {
        Some(self.get_mtp_debt_value(&mt, prices) / self.get_mtp_collateral_value(&mt, prices))
        }
    }
```

Listing 2.11: burrowland/contracts/contract/src/margin_position.rs

Impact The computation of the leverage rate is inaccurate.

Suggestion I Add the hold position fee into the computation of the leverage rate.

2.3 Additional Recommendation

2.3.1 Automatically construct swap indication from the token information

Status Confirmed

Introduced by Version 1

Description In functions internal_margin_open_position() and process_decrease_margin_position(), a consistency check is required between the token information and the swap_indication provided by the user within functions verify_token_in() and verify_token_out(). However, both of them are specified by the user.

```
pub(crate) fn internal_margin_open_position(
    &mut self,
    ts: Timestamp,
    account: &mut MarginAccount,
    token_c_id: &AccountId,
    token_c_amount: Balance,
    token_d_id: &AccountId,
    token_d_amount: Balance,
    token_p_id: &AccountId,
    min_token_p_amount: Balance,
    swap_indication: &SwapIndication,
    prices: &Prices,
) -> EventDataMarginOpen {
    let pos_id = format!("{}_{}", account.account_id.clone(), ts);
    assert!(
        !account.margin_positions.contains_key(&pos_id),
        "Margin position already exist"
    );

    let asset_c = self.internal_unwrap_asset(token_c_id);
    let asset_p = self.internal_unwrap_asset(token_p_id);
    let mut asset_d = self.internal_unwrap_asset(token_d_id);
    let margin_config = self.internal_margin_config();

    // check legitimacy: assets legal; swap_indication matches;
    margin_config.check_pair(&token_d_id, &token_p_id, &token_c_id);
    let mut swap_detail = self.parse_swap_indication(swap_indication);
    let ft_d_amount = token_d_amount / 10u128.pow(asset_d.config.extra_decimals as u32);
    assert!(
        swap_detail.verify_token_in(token_d_id, ft_d_amount),
        "token_in check failed"
    );
    let ft_p_amount =
        min_token_p_amount / 10u128.pow(asset_p.config.extra_decimals as u32);
    assert!(
        swap_detail.verify_token_out(token_p_id, ft_p_amount),
        "token_out check failed"
    );

    // check safty:
    //   min_position_amount reasonable
    assert!(
        is_min_amount_out_reasonable(
            token_d_amount,
            &asset_d,
            prices.get_unwrap(&token_d_id),
            &asset_p,
            prices.get_unwrap(&token_p_id),
            min_token_p_amount,
            margin_config.max_slippage_rate,
        ),
        "min_position_amount is too low"
    );
    //   margin_hf more than 1 + safty_buffer_rate(10%)
    let mut mt = MarginTradingPosition::new(
        ts,
        token_c_id.clone(),
        asset_c.supplied.amount_to_shares(token_c_amount, false),
        token_d_id.clone(),
        token_p_id.clone(),
    );
    mt.token_d_shares = asset_d.margin_debt.amount_to_shares(token_d_amount, true);
    mt.token_p_amount = min_token_p_amount;
    assert!(
        !self.is_mt_liquidatable(&mt, prices, margin_config.min_safty_buffer),
        "Debt is too much"
    );
    assert!(
        !self.is_mt_forcecloseable(&mt, prices),
        "Debt is too much"
    );
    //   leverage rate less than max leverage rate
    assert!(
        self.get_mtp_lr(&mt, prices).unwrap()
            <= BigDecimal::from(margin_config.max_leverage_rate as u32),
        "Leverage rate is too high"
    );

    // passes all check, start to open
    let event = EventDataMarginOpen {
        account_id: account.account_id.clone(),
        pos_id: pos_id.clone(),
        token_c_id: token_c_id.clone(),
        token_c_amount,
        token_c_shares: mt.token_c_shares,
        token_d_id: token_d_id.clone(),
        token_d_amount,
        token_p_id: token_p_id.clone(),
        token_p_amount: min_token_p_amount,
    };
    account.withdraw_supply_shares(token_c_id, &mt.token_c_shares);
    mt.token_d_shares.0 = 0;
    mt.token_p_amount = 0;
    asset_d.increase_margin_pending_debt(token_d_amount, margin_config.pending_debt_scale);
    self.internal_set_asset(token_d_id, asset_d);
    // TODO: may need to change to store in an unorderedmap in user Account
    account.margin_positions.insert(pos_id.clone(), mt);

    // step 4: call dex to trade and wait for callback
    // organize swap action
    let swap_ref = SwapReference {
        account_id: account.account_id.clone(),
        pos_id: pos_id.clone(),
        amount_in: token_d_amount.into(),
        op: format!("open"),
        liquidator_id: None,
    };
    swap_detail.set_client_echo(&swap_ref.to_msg_string());
    let swap_msg = swap_detail.to_msg_string();
    ext_fungible_token::ext(token_d_id.clone())
        .with_attached_deposit(1)
        .with_static_gas(GAS_FOR_FT_TRANSFER_CALL)
        .ft_transfer_call(
            swap_indication.dex_id.clone(),
            U128(ft_d_amount),
            None,
            swap_msg,
        )
        .then(
            Self::ext(env::current_account_id())
                .with_static_gas(GAS_FOR_FT_TRANSFER_CALL_CALLBACK)
                .callback_dex_trade(
                    account.account_id.clone(),
                    pos_id.clone(),
                    token_d_amount.into(),
                    U128(0),
                    format!("open"),
                ),
        );
        event
}
```

Listing 2.12: burrowland/contracts/contract/src/margin_position.rs

```
pub(crate) fn process_decrease_margin_position(
    &mut self,
    account: &mut MarginAccount,
    pos_id: &String,
    token_p_amount: Balance,
    min_token_d_amount: Balance,
    swap_indication: &SwapIndication,
    prices: &Prices,
    op: String,
    liquidator_id: Option<AccountId>,
) -> EventDataMarginDecrease {
    let mut mt = account
        .margin_positions
        .get_mut(pos_id)
        .expect("Position not exist");
    assert!(
        !mt.is_locking,
        "Position is currently waiting for a trading result."
    );
    let pre_token_p_amount = mt.token_p_amount;
    let mut asset_p = self.internal_unwrap_asset(&mt.token_p_id);
    let asset_d = self.internal_unwrap_asset(&mt.token_d_id);
    let margin_config = self.internal_margin_config();

    //  check swap_indication
    let mut swap_detail = self.parse_swap_indication(swap_indication);
    let ft_p_amount =
        token_p_amount / 10u128.pow(asset_p.config.extra_decimals as u32);
    assert!(
        swap_detail.verify_token_in(&mt.token_p_id, ft_p_amount),
        "token_in check failed"
    );
    let ft_d_amount = min_token_d_amount / 10u128.pow(asset_d.config.extra_decimals as u32);
    assert!(
        swap_detail.verify_token_out(&mt.token_d_id, ft_d_amount),
        "token_out check failed"
    );

    //  min_debt_amount reasonable
    assert!(
        is_min_amount_out_reasonable(
            token_p_amount,
            &asset_p,
            prices.get_unwrap(&mt.token_p_id),
            &asset_d,
            prices.get_unwrap(&mt.token_d_id),
            min_token_d_amount,
            margin_config.max_slippage_rate,
        ),
        "min_debt_amount is too low"
    );

    if op == "close" || op == "liquidate" {
        //  ensure all debt would be repaid
        //  and take holding-position fee into account
        let total_debt_amount = asset_d
```

```
                .margin_debt
                .shares_to_amount(mt.token_d_shares, true);
            let hp_fee = u128_ratio(
                mt.debt_cap,
                asset_d.unit_acc_hp_interest - mt.uahpi_at_open,
                UNIT,
            );
            if min_token_d_amount < total_debt_amount + hp_fee {
                assert_eq!(
                    mt.token_c_id, mt.token_d_id,
                    "Can NOT trade under total debt when margin and debt asset are not the same"
                );
                let gap_shares = asset_d
                    .supplied
                    .amount_to_shares(total_debt_amount + hp_fee - min_token_d_amount, true);
                assert!(
                    mt.token_c_shares.0 > gap_shares.0,
                    "Not all debt could be repaid"
                );
            }
        }

        if op == "liquidate" {
            assert!(
                self.is_mt_liquidatable(&mt, prices, margin_config.min_safty_buffer),
                "Margin position is not liquidatable"
            );
        } else if op == "forceclose" {
            assert!(
                self.is_mt_forcecloseable(&mt, prices),
                "Margin position is not forceclose-able"
            );
        }

        //   ensure enough position token to trade
        if token_p_amount > mt.token_p_amount {
            // try to add some of margin asset into trading
            assert_eq!(
                mt.token_c_id, mt.token_p_id,
                "Not enough position asset balance"
            );
            let gap_shares = asset_p
                .supplied
                .amount_to_shares(token_p_amount - mt.token_p_amount, true);
            mt.token_c_shares
                .0
                .checked_sub(gap_shares.0)
                .expect("Not enough position asset balance");
            asset_p
                .supplied
                .withdraw(gap_shares, token_p_amount - mt.token_p_amount);
            asset_p.margin_position -= mt.token_p_amount;
            mt.token_p_amount = 0;
        } else {
            asset_p.margin_position -= token_p_amount;
            mt.token_p_amount -= token_p_amount;
        }

        // prepare to close
        mt.is_locking = true;
        self.internal_set_asset(&mt.token_p_id, asset_p);
        // TODO: mt may be needed to change to store in an unorderedmap in user Account

        let event = EventDataMarginDecrease {
            account_id: account.account_id.clone(),
            pos_id: pos_id.clone(),
            liquidator_id: liquidator_id.clone(),
            token_p_id: mt.token_p_id.clone(),
            token_p_amount,
            token_d_id: mt.token_d_id.clone(),
            token_d_amount: min_token_d_amount,
        };

        // step 3: call dex to trade and wait for callback
        // organize swap action
        let swap_ref = SwapReference {
            account_id: account.account_id.clone(),
            pos_id: pos_id.clone(),
            amount_in: token_p_amount.into(),
            op,
            liquidator_id,
        };
        swap_detail.set_client_echo(&swap_ref.to_msg_string());
        let swap_msg = swap_detail.to_msg_string();
        ext_fungible_token::ext(mt.token_p_id.clone())
            .with_attached_deposit(1)
            .with_static_gas(GAS_FOR_FT_TRANSFER_CALL)
            .ft_transfer_call(
                swap_indication.dex_id.clone(),
                U128(ft_p_amount),
                None,
                swap_msg,
            )
            .then(
                Self::ext(env::current_account_id())
                    .with_static_gas(GAS_FOR_FT_TRANSFER_CALL_CALLBACK)
                    .callback_dex_trade(
                        account.account_id.clone(),
                        pos_id.clone(),
                        token_p_amount.into(),
                        pre_token_p_amount.into(),
                        format!("decrease"),
                    ),
            );
        event
    }
```

Listing 2.13: burrowland/contracts/contract/src/margin_position.rs

Suggestion I The contract can automatically construct the `swap_indication` based on the token information provided by the user.

Feedback from the Project The code will be optimized as appropriate in subsequent versions.

2.3.2 Use UnorderedMap for margin_positions instead of HashMap

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The current implementation uses a `HashMap` for the `margin_positions` field in the struct `MarginAccount`. This means that every time the `MarginAccount` is deserialized, all the `margin_positions` have to be deserialized as well, resulting in a waste of gas.

```rust
#[derive(BorshSerialize, BorshDeserialize, Serialize, Clone)]
#[serde(crate = "near_sdk::serde")]
pub struct MarginAccount {
   /// A copy of an account ID. Saves one storage_read when iterating on accounts.
   pub account_id: AccountId,
   /// A list of assets that are supplied by the account (but not used a collateral).
   /// It's not returned for account pagination.
   pub supplied: HashMap<TokenId, Shares>,
   // margin trading related
   pub margin_positions: HashMap<PosId, MarginTradingPosition>,
   /// Tracks changes in storage usage by persistent collections in this account.
   #[borsh_skip]
   #[serde(skip)]
   pub storage_tracker: StorageTracker,
}
```

Listing 2.14: burrowland/contracts/contract/src/margin_accounts.rs

Suggestion I Use `UnorderedMap` instead of `HashMap` for the `margin_positions` field in the struct `MarginAccount`.

2.3.3 Incorrect error message in get_token_out()

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `RefV2TokenReceiverMessage::get_token_out()`, "`RefV1TokenReceiverMessage`" is used as an error message, which is incorrect .

```rust
pub fn get_token_out(&self) -> (AccountId, Balance, Option<bool>) {
    if let RefV2TokenReceiverMessage::Swap {
        pool_ids: _,
        output_token,
        min_output_amount,
        skip_unwrap_near,
        client_echo: _,
    } = self
    {
        (
            output_token.clone(),
            min_output_amount.0,
            skip_unwrap_near.clone(),
        )
    } else {
        env::panic_str("Invalid RefV1TokenReceiverMessage");
    }
}
```

Listing 2.15: burrowland/contracts/contract/src/margin_trading.rs

Suggestion I Revise “`RefV1TokenReceiverMessage`” to “`RefV2TokenReceiverMessage`”.

BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS