Security Audit Report for BurrowLand

Date: Oct 24, 2023

Version: 1.0

Contact: contact@blocksec.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2023.10.24 19:34:19 +08'00'

Contents

1 Introduction **1**

1.1 About Target Contracts 1

1.2 Disclaimer 2

1.3 Procedure of Auditing 2

1.3.1 Software Security 2

1.3.2 DeFi Security 3

1.3.3 NFT Security 3

1.3.4 Additional Recommendation 3

1.4 Security Model 3

2 Findings **5**

2.1 DeFi Security 5

2.1.1 Improper Round Direction 5

2.1.2 Interactions Required for Newly Added Rewards 6

2.1.3 Precision Loss during Token Transfer 9

2.1.4 Lack of Updating Affected Farm 11

2.1.5 Inconsistency of Modifiable booster_token_id and booster_decimals 11

2.2 Additional Recommendation 15

2.2.1 Improper Discount Value Check 15

2.2.2 Redundant Invocation of add_affected_farm() 18

Report Manifest

Item	Description
Client	Ref Labs
Target	BurrowLand

Version History

Version	Date	Description
1.0	October 24, 2023	First Version

Chapter 1 Introduction

1.1 About Target Contracts

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

The repository that has been audited includes burrowland [1].

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

Project		Commit SHA
BurrowLand	`Version 1`	`5eb851cf361ce53e460ab8d5bd4a265487df5993`
	`Version 2`	`7b406f499cebb0d7820d46aa61e0751b95ef80e5`

Note that, we did **NOT** audit all the modules in the repository. The modules covered by this audit report include **burrowland/contract/src** folder contract only. Specifically, the files covered in this audit include:

- account.rs
- account_asset.rs
- account_farm.rs
- account_view.rs
- actions.rs
- asset.rs
- asset_config.rs
- asset_farm.rs
- asset_view.rs
- big_decimal.rs
- booster_staking.rs
- config.rs
- events.rs
- fungible_token.rs
- legacy.rs
- lib.rs
- pool.rs
- price_receiver.rs
- prices.rs
- storage.rs

[1]https://github.com/burrowHQ/burrowland/pull/1/files

  - storage_tracker.rs
  - upgrade.rs
  - utils.rs

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
* Access control
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

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

1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

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

1.4 Security Model

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

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

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.

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

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

Table 1.1: Vulnerability Severity Classification

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

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.

Chapter 2 Findings

In total, we find **five** potential issues. Besides, we have **two** recommendations as follows:

- High Risk: 0
- Medium Risk: 4
- Low Risk: 1
- Recommendations: 2
- Notes: 0

ID	Severity	Description	Category	Status
1	Low	Improper Round Direction	Defi Security	Fixed
2	Medium	Interactions Required for Newly Added Rewards	DeFi Security	Confirmed
3	Medium	Precision Loss during Token Transfer	DeFi Security	Confirmed
4	Medium	Lack of Updating Affected Farm	DeFi Security	Confirmed
5	Medium	Inconsistency of Modifiable booster_token_id and booster_decimals	DeFi Security	Fixed
6	-	Improper Discount Value Check	Recommendation	Confirmed
7	-	Redundant Invocation of add_affected_farm()	Recommendation	Fixed

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Improper Round Direction

Severity Low

Status Fixed in Version 2

Introduced by Version 1

Description The function asset_amount_to_shares() is used to compute an asset's shares and corresponding amount. When parameter inverse_round_direction is set to true, the calculation of shares based on amount will be rounded down, while the calculation of amount based on shares will be rounded up, and vice versa.

However, in function internal_borrow(), when calculating borrowed_shares and amount, the inverse_round_direction is set to true, which may cause the borrowed_shares to end up being smaller than expected.

```
pub fn internal_borrow(
    &mut self,
    account: &mut Account,
    asset_amount: &AssetAmount,
) -> Balance {
    let mut asset = self.internal_unwrap_asset(&asset_amount.token_id);
    assert!(asset.config.can_borrow, "Thi asset can't be used borrowed");

    let mut account_asset = account.internal_get_asset_or_default(&asset_amount.token_id);
```

```rust

    let available_amount = asset.available_amount();
    let max_borrow_shares = asset.borrowed.amount_to_shares(available_amount, false);

    let (borrowed_shares, amount) =
        asset_amount_to_shares(&asset.borrowed, max_borrow_shares, &asset_amount, true);

    assert!(
        amount <= available_amount,
        "Borrow error: Exceeded available amount {} of {}",
        available_amount,
        &asset_amount.token_id
    );

    let supplied_shares: Shares = asset.supplied.amount_to_shares(amount, false);

    asset.borrowed.deposit(borrowed_shares, amount);
    asset.supplied.deposit(supplied_shares, amount);
    self.internal_set_asset(&asset_amount.token_id, asset);

    account.increase_borrowed(&asset_amount.token_id, borrowed_shares);

    account_asset.deposit_shares(supplied_shares);
    account.internal_set_asset(&asset_amount.token_id, account_asset);

    amount
}
```

Listing 2.1: actions.rs

Impact Improper rounding may result in the contract recording a slightly lower share for a user's borrowed amount than expected.

Suggestion Change the arguments `inverse_round_direction` of this invocation to `false`.

2.1.2 Interactions Required for Newly Added Rewards

Severity Medium

Status Confirmed

Introduced by Version 1

Description The function `add_asset_farm_reward()` is used to add rewards to a farm. If a new reward token is added for a farm, users cannot receive the subsequent rewards until the function `internal_account_farm_claim()` is invoked even if the user already holds the corresponding token.

```rust
#[payable]
pub fn add_asset_farm_reward(
    &mut self,
    farm_id: FarmId,
    reward_token_id: AccountId,
    new_reward_per_day: U128,
    new_booster_log_base: U128,
    reward_amount: U128,
) {
    assert_one_yocto();
    self.assert_owner();
    match &farm_id {
        FarmId::Supplied(token_id) | FarmId::Borrowed(token_id) => {
            assert!(self.assets.contains_key(token_id));
        }
        FarmId::NetTvl => {}
    };
    let reward_token_id: TokenId = reward_token_id.into();
    let mut reward_asset = self.internal_unwrap_asset(&reward_token_id);
    assert!(
        reward_asset.reserved >= reward_amount.0
            && reward_asset.available_amount() >= reward_amount.0,
        "Not enough reserved reward balance"
    );
    reward_asset.reserved -= reward_amount.0;
    self.internal_set_asset(&reward_token_id, reward_asset);
    let mut asset_farm = self
        .internal_get_asset_farm(&farm_id, false)
        .unwrap_or_else(|| AssetFarm {
            block_timestamp: env::block_timestamp(),
            rewards: HashMap::new(),
            inactive_rewards: LookupMap::new(StorageKey::InactiveAssetFarmRewards {
                farm_id: farm_id.clone(),
            }),
        });

    let mut asset_farm_reward = asset_farm
        .rewards
        .remove(&reward_token_id)
        .or_else(|| asset_farm.internal_remove_inactive_asset_farm_reward(&reward_token_id))
        .unwrap_or_default();
    asset_farm_reward.reward_per_day = new_reward_per_day.into();
    asset_farm_reward.booster_log_base = new_booster_log_base.into();
    asset_farm_reward.remaining_rewards += reward_amount.0;
    asset_farm
        .rewards
        .insert(reward_token_id, asset_farm_reward);
    self.internal_set_asset_farm(&farm_id, asset_farm);
}
```

Listing 2.2: config.rs

```
    pub fn internal_account_farm_claim(
        &self,
        account: &Account,
        farm_id: &FarmId,
        asset_farm: &AssetFarm,
    ) -> (
        AccountFarm,
        Vec<(TokenId, Balance)>,
        Vec<(TokenId, Balance)>,
    ) {
        let mut new_rewards = vec![];
        let mut inactive_rewards = vec![];
        let block_timestamp = env::block_timestamp();
        let mut account_farm: AccountFarm = account
            .farms
            .get(farm_id)
            .cloned()
            .unwrap_or_else(AccountFarm::new);
        if account_farm.block_timestamp != block_timestamp {
            account_farm.block_timestamp = block_timestamp;
            let mut old_rewards = std::mem::take(&mut account_farm.rewards);
            for (
                token_id,
                AssetFarmReward {
                    reward_per_share, ..
                },
            ) in &asset_farm.rewards
            {
                let boosted_shares = if let Some(AccountFarmReward {
                    boosted_shares,
                    last_reward_per_share,
                }) = old_rewards.remove(token_id)
                {
                    let diff = reward_per_share.clone() - last_reward_per_share;
                    let amount = diff.round_mul_u128(boosted_shares);
                    if amount > 0 {
                        new_rewards.push((token_id.clone(), amount));
                    }
                    boosted_shares
                } else {
                    0
                };
                account_farm.rewards.insert(
                    token_id.clone(),
                    AccountFarmReward {
                        boosted_shares,
                        last_reward_per_share: reward_per_share.clone(),
                    },
                );
            }
            for (
                token_id,
                AccountFarmReward {
                    boosted_shares,
                    last_reward_per_share,
                },
            ) in old_rewards
            {
                let AssetFarmReward {
                    reward_per_share, ..
                } = asset_farm
                    .internal_get_inactive_asset_farm_reward(&token_id)
                    .unwrap();
                let diff = reward_per_share - last_reward_per_share;
                let amount = diff.round_mul_u128(boosted_shares);
                inactive_rewards.push((token_id.clone(), boosted_shares));
                if amount > 0 {
                    new_rewards.push((token_id, amount));
                }
            }
        }
        (account_farm, new_rewards, inactive_rewards)
    }
```

Listing 2.3: account_farm.rs

Impact Users may receive less rewards than expected.

Suggestion Ensure the rewards can be accumulated since it's been added.

2.1.3 Precision Loss during Token Transfer

Severity Medium

Status Confirmed

Introduced by Version 1

Description The function internal_ft_transfer() is used to transfer funds to other accounts. Tokens can have different decimals configured in their specifications. To standardize transfer amounts, any input value is first converted based on the token's extra_decimals property. Specifically, the actual transferred amount (i.e., ft_amount) is derived by dividing the input by extra_decimals. However, precision loss during this calculation process is not taken into account, which may cause losses for users.

Similar problems also can be found in functions internal_ft_transfer_prot_own(), and internal_ft_transfer_reserved().

```
 pub fn internal_ft_transfer(
    &mut self,
    account_id: &AccountId,
    token_id: &TokenId,
    amount: Balance,
) -> Promise {
    let asset = self.internal_unwrap_asset(token_id);
    let ft_amount = amount / 10u128.pow(asset.config.extra_decimals as u32);
    ext_fungible_token::ft_transfer(
        account_id.clone(),
        ft_amount.into(),
        None,
        token_id.clone(),
        ONE_YOCTO,
        GAS_FOR_FT_TRANSFER,
    )
    .then(ext_self::after_ft_transfer(
        account_id.clone(),
        token_id.clone(),
        amount.into(),
            env::current_account_id(),
            NO_DEPOSIT,
            GAS_FOR_AFTER_FT_TRANSFER,
        ))
    }

    pub fn internal_ft_transfer_prot_own(
        &mut self,
        account_id: &AccountId,
        token_id: &TokenId,
        amount: Balance,
        stdd_amount: Balance,
    ) -> Promise {
        ext_fungible_token::ft_transfer(
            account_id.clone(),
            amount.into(),
            None,
            token_id.clone(),
            ONE_YOCTO,
            GAS_FOR_FT_TRANSFER,
        )
        .then(ext_self::after_ft_transfer_prot_own(
            account_id.clone(),
            token_id.clone(),
            stdd_amount.into(),
            env::current_account_id(),
            NO_DEPOSIT,
            GAS_FOR_AFTER_FT_TRANSFER_PROT_OWN,
        ))
    }

    pub fn internal_ft_transfer_reserved(
        &mut self,
        account_id: &AccountId,
        token_id: &TokenId,
        amount: Balance,
        stdd_amount: Balance,
    ) -> Promise {
        ext_fungible_token::ft_transfer(
            account_id.clone(),
            amount.into(),
            None,
            token_id.clone(),
            ONE_YOCTO,
            GAS_FOR_FT_TRANSFER,
        )
        .then(ext_self::after_ft_transfer_reserved(
            account_id.clone(),
            token_id.clone(),
            stdd_amount.into(),
            env::current_account_id(),
            NO_DEPOSIT,
            GAS_FOR_AFTER_FT_TRANSFER_RESERVED,
        ))
    }
```

Listing 2.4: fungible_token.rs

Impact Users may receive less tokens from the contract than expected.

Suggestion Handle the precision loss properly.

2.1.4 Lack of Updating Affected Farm

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description Function `after_ft_transfer()` is used to handle the potential transfer failures. When the transfer fails, the function will redeposit the tokens in the amount that was previously deducted back to the user's account. However, after the deposit, the function does not invoke the function `internal_account_apply_affected_farms()` to update the affected farms of the user.

```rust
    #[private]
    fn after_ft_transfer(
        &mut self,
        account_id: AccountId,
        token_id: TokenId,
        amount: U128,
    ) -> bool {
        let promise_success = is_promise_success();
        if !promise_success {
            let mut account = self.internal_unwrap_account(&account_id);
            account.add_affected_farm(FarmId::Supplied(token_id.clone()));
            self.internal_deposit(&mut account, &token_id, amount.0);
            events::emit::withdraw_failed(&account_id, amount.0, &token_id);
            self.internal_set_account(&account_id, account);
        } else {
            events::emit::withdraw_succeeded(&account_id, amount.0, &token_id);
        }
        promise_success
    }
```

Listing 2.5: fungible_token.rs

Impact The redeposit tokens are not included in farming, resulting in the user's rewards to be less than expected.

Suggestion Invoke the function `internal_account_apply_affected_farms()` to update the account affected farm in function `after_ft_transfer()`.

2.1.5 Inconsistency of Modifiable booster_token_id and booster_decimals

Severity Medium

Status Fixed in `Version 2`

Introduced by Version 1

Description According to the design of protocol, users can deposit a specific token (i.e., booster_token_id) to boost the farming process. The token contract address and decimals are configured in the struct Config. However, the owner has the authority to reconfigure the values of each element in Config via the function update_config(). If the booster_token_id and decimal are updated during the user's staking process and the user invokes the function account_unstake_booster() to unstake, the actual token withdrawn would be different from the original token deposited.

```rust
pub struct Config {
/// The account ID of the oracle contract
pub oracle_account_id: AccountId,

/// The account ID of the contract owner that allows to modify config, assets and use reserves
    .
pub owner_id: AccountId,

/// The account ID of the booster token contract.
pub booster_token_id: TokenId,

/// The number of decimals of the booster fungible token.
pub booster_decimals: u8,

/// The total number of different assets
pub max_num_assets: u32,

/// The maximum number of seconds expected from the oracle price call.
pub maximum_recency_duration_sec: DurationSec,

/// Maximum staleness duration of the price data timestamp.
/// Because NEAR protocol doesn't implement the gas auction right now, the only reason to
/// delay the price updates are due to the shard congestion.
/// This parameter can be updated in the future by the owner.
pub maximum_staleness_duration_sec: DurationSec,

/// The minimum duration to stake booster token in seconds.
pub minimum_staking_duration_sec: DurationSec,

/// The maximum duration to stake booster token in seconds.
pub maximum_staking_duration_sec: DurationSec,

/// The rate of xBooster for the amount of Booster given for the maximum staking duration.
/// Assuming the 100% multiplier at the minimum staking duration. Should be no less than 100%.
/// E.g. 20000 means 200% multiplier (or 2X).
pub x_booster_multiplier_at_maximum_staking_duration: u32,

/// Whether an account with bad debt can be liquidated using reserves.
/// The account should have borrowed sum larger than the collateral sum.
pub force_closing_enabled: bool,
}
```

Listing 2.6: config.rs

```rust
pub fn update_config(&mut self, config: Config) {
    assert_one_yocto();
    self.assert_owner();
    config.assert_valid();
    self.config.set(&config);
}
```

Listing 2.7: config.rs

```rust
#[payable]
pub fn account_stake_booster(&mut self, amount: Option<U128>, duration: DurationSec) {
    assert_one_yocto();
    let config = self.internal_config();

    assert!(
        duration >= config.minimum_staking_duration_sec
            && duration <= config.maximum_staking_duration_sec,
        "Duration is out of range"
    );

    let account_id = env::predecessor_account_id();
    let mut account = self.internal_unwrap_account(&account_id);

    let booster_token_id = config.booster_token_id.clone();

    // Computing and withdrawing amount from supplied.
    let mut asset = self.internal_unwrap_asset(&booster_token_id);
    let mut account_asset = account.internal_unwrap_asset(&booster_token_id);

    let (shares, amount) = if let Some(amount) = amount.map(|a| a.0) {
        (asset.supplied.amount_to_shares(amount, true), amount)
    } else {
        (
            account_asset.shares,
            asset.supplied.shares_to_amount(account_asset.shares, false),
        )
    };
    assert!(
        shares.0 > 0 && amount > 0,
        "The amount should be greater than zero"
    );

    account_asset.withdraw_shares(shares);
    account.internal_set_asset(&booster_token_id, account_asset);

    asset.supplied.withdraw(shares, amount);
    self.internal_set_asset(&booster_token_id, asset);

    // Computing amount of the new xBooster token and new unlock timestamp.
    let timestamp = env::block_timestamp();
    let new_duration_ns = sec_to_nano(duration);
    let new_unlock_timestamp_ns = timestamp + new_duration_ns;

```

```rust
        let mut booster_staking = account
            .booster_staking
            .take()
            .map(|mut booster_staking| {
                assert!(
                    booster_staking.unlock_timestamp <= new_unlock_timestamp_ns,
                    "The new staking duration is shorter than the current remaining staking duration"
                );
                let restaked_x_booster_amount = compute_x_booster_amount(
                    &config,
                    booster_staking.staked_booster_amount,
                    new_duration_ns,
                );
                booster_staking.x_booster_amount =
                    std::cmp::max(booster_staking.x_booster_amount, restaked_x_booster_amount);
                booster_staking
            })
            .unwrap_or_default();
        booster_staking.unlock_timestamp = new_unlock_timestamp_ns;
        booster_staking.staked_booster_amount += amount;
        let extra_x_booster_amount = compute_x_booster_amount(&config, amount, new_duration_ns);
        booster_staking.x_booster_amount += extra_x_booster_amount;

        events::emit::booster_stake(
            &account_id,
            amount,
            duration,
            extra_x_booster_amount,
            &booster_staking,
        );

        account.booster_staking.replace(booster_staking);

        account
            .affected_farms
            .extend(account.get_all_potential_farms());
        account.add_affected_farm(FarmId::Supplied(config.booster_token_id.clone()));
        self.internal_account_apply_affected_farms(&mut account);
        self.internal_set_account(&account_id, account);
    }

    #[payable]
    pub fn account_unstake_booster(&mut self) {
        assert_one_yocto();

        let config = self.internal_config();
        let account_id = env::predecessor_account_id();
        let mut account = self.internal_unwrap_account(&account_id);

        let timestamp = env::block_timestamp();
        let booster_staking = account
            .booster_staking
            .take()
            .expect("No staked booster token");
        assert!(
            booster_staking.unlock_timestamp <= timestamp,
            "The staking is not unlocked yet"
        );

        self.internal_deposit(
            &mut account,
            &config.booster_token_id,
            booster_staking.staked_booster_amount,
        );

        events::emit::booster_unstake(&account_id, &booster_staking);

        account
            .affected_farms
            .extend(account.get_all_potential_farms());
        self.internal_account_apply_affected_farms(&mut account);
        self.internal_set_account(&account_id, account);
    }
}
```

Listing 2.8: booster_staking.rs

Impact Users may withdraw tokens that are different from their original deposited ones.

Suggestion Make `booster_token_id` and `decimal` immutable after the initial configuration.

2.2 Additional Recommendation

2.2.1 Improper Discount Value Check

Status Confirmed

Introduced by `Version 1`

Description The function `internal_liquidate()` is used to handle liquidation logic of the protocol. The liquidated account's health factor is calculated by the function `compute_max_discount()`. However, the function `internal_liquidate()` requires that after liquidation, the liquidated account's should stay in risk (lines 371 - 374), which means the liquidator can never fully liquidate all the assets of that account. This does not align with the protocol's intention.

```
307     pub fn internal_liquidate(
308         &mut self,
309         account_id: &AccountId,
310         account: &mut Account,
311         prices: &Prices,
312         liquidation_account_id: &AccountId,
313         in_assets: Vec<AssetAmount>,
314         out_assets: Vec<AssetAmount>,
315     ) {
316         let mut liquidation_account = self.internal_unwrap_account(liquidation_account_id);
```

```

        let max_discount = self.compute_max_discount(&liquidation_account, &prices);
        assert!(
            max_discount > BigDecimal::zero(),
            "The liquidation account is not at risk"
        );

        let mut borrowed_repaid_sum = BigDecimal::zero();
        let mut collateral_taken_sum = BigDecimal::zero();

        for asset_amount in in_assets {
            liquidation_account.add_affected_farm(FarmId::Borrowed(asset_amount.token_id.clone()));
            let mut account_asset = account.internal_unwrap_asset(&asset_amount.token_id);
            let amount =
                self.internal_repay(&mut account_asset, &mut liquidation_account, &asset_amount);
            account.internal_set_asset(&asset_amount.token_id, account_asset);
            let asset = self.internal_unwrap_asset(&asset_amount.token_id);

            borrowed_repaid_sum = borrowed_repaid_sum
                + BigDecimal::from_balance_price(
                    amount,
                    prices.get_unwrap(&asset_amount.token_id),
                    asset.config.extra_decimals,
                );
        }

        for asset_amount in out_assets {
            let asset = self.internal_unwrap_asset(&asset_amount.token_id);
            liquidation_account.add_affected_farm(FarmId::Supplied(asset_amount.token_id.clone()));
            let mut account_asset = account.internal_get_asset_or_default(&asset_amount.token_id);
            let amount = self.internal_decrease_collateral(
                &mut account_asset,
                &mut liquidation_account,
                &asset_amount,
            );
            account.internal_set_asset(&asset_amount.token_id, account_asset);

            collateral_taken_sum = collateral_taken_sum
                + BigDecimal::from_balance_price(
                    amount,
                    prices.get_unwrap(&asset_amount.token_id),
                    asset.config.extra_decimals,
                );
        }

        let discounted_collateral_taken = collateral_taken_sum * (BigDecimal::one() - max_discount)
            ;
        assert!(
            discounted_collateral_taken <= borrowed_repaid_sum,
            "Not enough balances repaid: discounted collateral {} > borrowed repaid sum {}",
            discounted_collateral_taken,
            borrowed_repaid_sum
        );

        let new_max_discount = self.compute_max_discount(&liquidation_account, &prices);
        assert!(
            new_max_discount > BigDecimal::zero(),
            "The liquidation amount is too large. The liquidation account should stay in risk"
        );
        assert!(
            new_max_discount < max_discount,
            "The health factor of liquidation account can't decrease. New discount {} < old
                discount {}",
            new_max_discount, max_discount
        );

        self.internal_account_apply_affected_farms(&mut liquidation_account);
        self.internal_set_account(liquidation_account_id, liquidation_account);

        events::emit::liquidate(
            &account_id,
            &liquidation_account_id,
            &collateral_taken_sum,
            &borrowed_repaid_sum,
        );
    }
```

Listing 2.9: actions.rs

```
    pub fn compute_max_discount(&self, account: &Account, prices: &Prices) -> BigDecimal {
        if account.borrowed.is_empty() {
            return BigDecimal::zero();
        }

        let collateral_sum =
            account
                .collateral
                .iter()
                .fold(BigDecimal::zero(), |sum, (token_id, shares)| {
                    let asset = self.internal_unwrap_asset(&token_id);
                    let balance = asset.supplied.shares_to_amount(*shares, false);
                    sum + BigDecimal::from_balance_price(
                        balance,
                        prices.get_unwrap(&token_id),
                        asset.config.extra_decimals,
                    )
                    .mul_ratio(asset.config.volatility_ratio)
                });

        let borrowed_sum =
            account
                .borrowed
                .iter()
                .fold(BigDecimal::zero(), |sum, (token_id, shares)| {
                    let asset = self.internal_unwrap_asset(&token_id);
                    let balance = asset.borrowed.shares_to_amount(*shares, true);
                    sum + BigDecimal::from_balance_price(
                        balance,
                        prices.get_unwrap(&token_id),
                        asset.config.extra_decimals,
                    )
                    .div_ratio(asset.config.volatility_ratio)
                });

        if borrowed_sum <= collateral_sum {
            BigDecimal::zero()
        } else {
            (borrowed_sum - collateral_sum) / borrowed_sum / BigDecimal::from(2u32)
        }
    }
```

Listing 2.10: actions.rs

Suggestion I Allows the liquidated account to be fully liquidated.

2.2.2 Redundant Invocation of add_affected_farm()

Status Fixed in Version 2

Introduced by Version 1

Description The usage of the function add_affected_farm() in multiple places within the contract is redundant. For example, in function ft_on_transfer(), when a user deposits tokens, the function records a type of Supplied FarmId for the user's account via the function add_affected_farm(). However, in the function internal_deposit(), add_affected_farm() is invoked again in the function internal_set_asset(), which is redundant.

```
    fn ft_on_transfer(
        &mut self,
        sender_id: AccountId,
        amount: U128,
        msg: String,
    ) -> PromiseOrValue<U128> {
        let token_id = env::predecessor_account_id();
        let mut asset = self.internal_unwrap_asset(&token_id);
        assert!(
            asset.config.can_deposit,
            "Deposits for this asset are not enabled"
        );

        let amount = amount.0 * 10u128.pow(asset.config.extra_decimals as u32);

        // TODO: We need to be careful that only whitelisted tokens can call this method with a
        //     given set of actions. Or verify which actions are possible to do.
        let actions: Vec<Action> = if msg.is_empty() {
            vec![]
        } else {
            let token_receiver_msg: TokenReceiverMsg =
                serde_json::from_str(&msg).expect("Can't parse TokenReceiverMsg");
        match token_receiver_msg {
            TokenReceiverMsg::Execute { actions } => actions,
            TokenReceiverMsg::DepositToReserve => {
                asset.reserved += amount;
                self.internal_set_asset(&token_id, asset);
                events::emit::deposit_to_reserve(&sender_id, amount, &token_id);
                return PromiseOrValue::Value(U128(0));
            }
        }
    };

    let mut account = self.internal_unwrap_account(&sender_id);
    account.add_affected_farm(FarmId::Supplied(token_id.clone()));
    self.internal_deposit(&mut account, &token_id, amount);
    events::emit::deposit(&sender_id, amount, &token_id);
    self.internal_execute(&sender_id, &mut account, actions, Prices::new());
    self.internal_set_account(&sender_id, account);

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

Listing 2.11: fungible_token.rs

```
pub fn internal_deposit(
    &mut self,
    account: &mut Account,
    token_id: &TokenId,
    amount: Balance,
) -> Shares {
    let mut asset = self.internal_unwrap_asset(token_id);
    let mut account_asset = account.internal_get_asset_or_default(token_id);

    let shares: Shares = asset.supplied.amount_to_shares(amount, false);

    account_asset.deposit_shares(shares);
    account.internal_set_asset(&token_id, account_asset);

    asset.supplied.deposit(shares, amount);
    self.internal_set_asset(token_id, asset);

    shares
}
```

Listing 2.12: actions.rs

```
pub fn internal_set_asset(&mut self, token_id: &TokenId, account_asset: AccountAsset) {
    if account_asset.is_empty() {
        self.supplied.remove(token_id);
    } else {
        self.supplied.insert(token_id.clone(), account_asset.shares);
    }
    self.add_affected_farm(FarmId::Supplied(token_id.clone()));
}
```

Listing 2.13: account_asset.rs

The following table contains all the redundant invocations of `add_affected_farm()`.

File	Line
actions.rs	51
actions.rs	76
actions.rs	83
actions.rs	345
booster_staking.rs	103
fungible_token.rs	60

Suggestion I Remove the redundant `add_affected_farm()` accordingly.