Security Audit Report for MemeFarming

Date: Feb 27, 2024

Version: 1.0

Contact: contact@blocksec.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2024.03.05 16:49:35 +08'00'

Contents

Report Manifest

Item	Description
Client	Ref-Finance
Target	MemeFarming

Version History

Version	Date	Description
1.0	February 27, 2024	First Version

Chapter 1 Introduction

1.1 About Target Contracts

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

The target of this audit is the code repository of MemeFarming[1] of Ref-Finance.

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

Project		Commit SHA
MemeFarming	`Version 1`	`8520aa714b8451fdbbc77d814359c86decfef555`
	`Version 2`	`b5c749a8938921dca625645c1ff7618db3aab718`

1.2 Disclaimer

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

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

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

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

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

[1]`https://github.com/ref-finance/boost-farm/tree/meme-farming`

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

We show the main concrete checkpoints in the following.

1.3.1 Software Security

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

1.3.2 DeFi Security

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

1.3.3 NFT Security

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

1.3.4 Additional Recommendation

∗ Gas optimization
∗ Code quality and style

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

1.4 Security Model

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

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

Table 1.1: Vulnerability Severity Classification

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

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

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

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

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

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

Chapter 2 Findings

In total, we find **five** potential issues. Besides, we also have **three** recommendations.

- High Risk: 1
- Medium Risk: 1
- Low Risk: 3
- Recommendations: 3
- Note: 0

ID	Severity	Description	Category	Status
1	High	Lack of Check in storage_unregister()	DeFi Security	Fixed
2	Low	Improper Decreasing Logic of farmer_count	DeFi Security	Fixed
3	Medium	Incorrect Timestamp in Roll Back Logic	DeFi Security	Confirmed
4	Low	Incorrect Calculation of decreased_seed_-power	DeFi Security	Fixed
5	Low	Lack of Duration Check	DeFi Security	Confirmed
6	-	Redundant Check in storage_withdraw()	Recommendation	Fixed
7	-	Lack of assert_one_yocto Check	Recommendation	Confirmed
8	-	Lack of Minimum Deposit Check	Recommendation	Fixed

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Lack of Check in storage_unregister()

Severity High

Status Fixed in Version 2

Introduced by Version 1

Description Function storage_unregister() is used for users to unregister their accounts and withdraw the previously deposited storage fee. It will check to ensure that there are no staked tokens or rewards remaining in the internal account before removing the account. However, it does not verify whether there are any tokens remaining in the withdraws field of the account.

```rust
#[allow(unused_variables)]
#[payable]
fn storage_unregister(&mut self, force: Option<bool>) -> bool {
    assert_one_yocto();
    require!(self.data().state == RunningState::Running, E004_CONTRACT_PAUSED);


    // force option is useless, leave it for compatible consideration.
    // User should withdraw all his rewards and seeds token before unregister!


    let account_id = env::predecessor_account_id();
    if let Some(farmer) = self.internal_get_farmer(&account_id) {

        require!(
            farmer.rewards.is_empty(),
            E103_STILL_HAS_REWARD
        );
        require!(
            farmer.seeds.is_empty(),
            E104_STILL_HAS_SEED
        );


        self.data_mut().farmers.remove(&account_id);
        self.data_mut().farmer_count -= 1;
        Promise::new(farmer.sponsor_id.clone()).transfer(STORAGE_BALANCE_MIN_BOUND);
        true
    } else {
        false
    }
}
```

Listing 2.1: storage_impl.rs

Impact Accounts' funds may not be withdrawn.

Suggestion Add a check to ensure that the `withdraws` field contains no funds in function `storage_unregister()`.

2.1.2 Improper Decreasing Logic of farmer_count

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The field `farmer_count` is used to count the number of non-empty `FarmerSeed` in the `Seed`. However, when removing a `FarmerSeed`, it does not consider whether it was originally empty or not. Since the function `unlock_and_unstake_seed()` allows the user to unlock/unstake 0 token, it's possible that the status of corresponding `farmer_seed` is originally empty. In this case, the `farmer_count` is incorrectly decreased by 1.

```
#[payable]
pub fn unlock_and_unstake_seed(
    &mut self,
    seed_id: SeedId,
    unlock_amount: U128,
    unstake_amount: U128,
) {
    assert_one_yocto();
    require!(self.data().state == RunningState::Running, E004_CONTRACT_PAUSED);


    let unlock_amount: Balance = unlock_amount.into();
    let unstake_amount: Balance = unstake_amount.into();


    let farmer_id = env::predecessor_account_id();


    let mut farmer = self.internal_unwrap_farmer(&farmer_id);
    let mut seed = self.internal_unwrap_seed(&seed_id);


    self.internal_do_farmer_claim(&mut farmer, &mut seed);


    let mut farmer_seed = farmer.seeds.get(&seed_id).unwrap();


    let prev = farmer_seed.get_seed_power();


    let decreased_seed_power =
    if unlock_amount > 0 {
        farmer_seed.unlock_to_free(unlock_amount)
    } else {
        0
    };
    if unstake_amount > 0 {
        farmer_seed.withdraw_free(unstake_amount);
        farmer.add_withdraw_seed(&seed_id, unstake_amount);
    }


    seed.total_seed_amount -= unstake_amount;
    seed.total_seed_power = seed.total_seed_power - prev + farmer_seed.get_seed_power();


    if farmer_seed.is_empty() {
        farmer.seeds.remove(&seed_id);
        if seed.farmer_count > 0 {
            seed.farmer_count -= 1;
        }
    } else {
        farmer.seeds.insert(&seed_id, &farmer_seed);
    }


    self.update_impacted_seeds(&mut farmer, &seed_id);


    self.internal_set_farmer(&farmer_id, farmer);
    self.internal_set_seed(&seed_id, seed);


    if unlock_amount > 0 {
        Event::SeedUnlock {
            farmer_id: &farmer_id,
            seed_id: &seed_id,
            unlock_amount: &U128(unlock_amount),
            decreased_power: &U128(decreased_seed_power),
            slashed_seed: &U128(0),
        }
        .emit();
    }
}
```

Listing 2.2: actions_of_farmer_seed.rs

Impact The count of `farmer`s within the `seed` is not accurate.

Suggestion Revise the corresponding logic, subtract 1 from `farmer_count` only if the `farmer_seed` to be removed was not empty before.

2.1.3 Incorrect Timestamp in Roll Back Logic

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description The function `callback_withdraw_seed()` is designed to handle the promise result of the operation of withdrawals. When the withdrawal fails, the status of the `farmer` will be rolled back in function `add_withdraw_seed()`. However, during the roll back process, the `apply_timestamp` is set as the current timestamp, which is incorrect.

```
 #[private]
pub fn callback_withdraw_seed(&mut self, seed_id: SeedId, sender_id: AccountId, amount: U128)
    -> bool {
    require!(
        env::promise_results_count() == 1,
        E001_PROMISE_RESULT_COUNT_INVALID
    );
    let amount: Balance = amount.into();
    match env::promise_result(0) {
        PromiseResult::NotReady => unreachable!(),
        PromiseResult::Failed => {
            // all seed amount goes back to withdraws
            if let Some(mut farmer) = self.internal_get_farmer(&sender_id) {
                farmer.add_withdraw_seed(&seed_id, amount);
                self.internal_set_farmer(&sender_id, farmer);
            } else {
                // if inner farmer not exist, goes to lostfound
                let seed_amount = self.data().seeds_lostfound.get(&seed_id).unwrap_or(0);
                self.data_mut()
                    .seeds_lostfound
                    .insert(&seed_id, &(seed_amount + amount));
            }
             Event::SeedWithdraw {
                farmer_id: &sender_id,
                seed_id: &seed_id,
                withdraw_amount: &U128(amount),
                success: false,
            }
            .emit();
            false
        }
        PromiseResult::Successful(_) => {
            Event::SeedWithdraw {
                farmer_id: &sender_id,
                seed_id: &seed_id,
                withdraw_amount: &U128(amount),
                success: true,
            }
            .emit();
            true
        }
    }
}
```

Listing 2.3: actions_of_farmer_seed.rs

```
pub fn add_withdraw_seed(&mut self, seed_id: &SeedId, amount: Balance) {
    if let Some(mut withdraw_seed) = self.withdraws.get_mut(seed_id) {
        withdraw_seed.amount += amount;
        withdraw_seed.apply_timestamp = env::block_timestamp();
    } else {
        self.withdraws.insert(seed_id.clone(), FarmerWithdraw {
            amount,
            apply_timestamp: env::block_timestamp(),
        });
    }
}
```

Listing 2.4: farmer.rs

Impact If the transfer fails, the user will have to wait for another `delay_withdraw_sec` duration before being able to withdraw again.

Suggestion Revise the corresponding logic.

Feedback This is by design.

2.1.4 Incorrect Calculation of decreased_seed_power

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In function `unlock_and_unstake_seed()`, the `decrease_seed_power` will be logged in the `event` to reflect how much “`seed power`” has been decreased by the user during the unlock/unstake operation. However, it only considers the impact of `unlocking` on “`seed power`”, but does not take into account the impact of `unstaking`.

```
 #[payable]
pub fn unlock_and_unstake_seed(
    &mut self,
    seed_id: SeedId,
    unlock_amount: U128,
    unstake_amount: U128,
) {
    assert_one_yocto();
    require!(self.data().state == RunningState::Running, E004_CONTRACT_PAUSED);


    let unlock_amount: Balance = unlock_amount.into();
    let unstake_amount: Balance = unstake_amount.into();


    let farmer_id = env::predecessor_account_id();


    let mut farmer = self.internal_unwrap_farmer(&farmer_id);
    let mut seed = self.internal_unwrap_seed(&seed_id);


    self.internal_do_farmer_claim(&mut farmer, &mut seed);


    let mut farmer_seed = farmer.seeds.get(&seed_id).unwrap();


    let prev = farmer_seed.get_seed_power();


    let decreased_seed_power =
    if unlock_amount > 0 {
        farmer_seed.unlock_to_free(unlock_amount)
    } else {
        0
    };
    if unstake_amount > 0 {
        farmer_seed.withdraw_free(unstake_amount);
        farmer.add_withdraw_seed(&seed_id, unstake_amount);
    }


    seed.total_seed_amount -= unstake_amount;
    seed.total_seed_power = seed.total_seed_power - prev + farmer_seed.get_seed_power();


    if farmer_seed.is_empty() {
        farmer.seeds.remove(&seed_id);
        if seed.farmer_count > 0 {
            seed.farmer_count -= 1;
        }
    } else {
        farmer.seeds.insert(&seed_id, &farmer_seed);
    }


    self.update_impacted_seeds(&mut farmer, &seed_id);


    self.internal_set_farmer(&farmer_id, farmer);
    self.internal_set_seed(&seed_id, seed);


    if unlock_amount > 0 {
        Event::SeedUnlock {
            farmer_id: &farmer_id,
            seed_id: &seed_id,
            unlock_amount: &U128(unlock_amount),
            decreased_power: &U128(decreased_seed_power),
            slashed_seed: &U128(0),
        }
        .emit();
    }
}
```

Listing 2.5: actions_of_farmer_seed.rs

Impact The decreased_power of the event SeedUnlock will be incorrect.

Suggestion Correctly calculate the decreased "seed power".

2.1.5 Lack of Duration Check

Severity Low

Status Confirmed

Introduced by Version 1

Description In function stake_lock_seed(), when locking a seed, it verifies that duration_sec is greater than or equal to seed.min_locking_duration_sec and less than or equal to config.maximum_locking_duration_sec. However, in the function modify_locking_policy(), there is no validation to ensure that config.maximum_locking_duration_sec is greater than the min_locking_duration_sec of all seeds.

```
 #[payable]
pub fn modify_locking_policy(&mut self, max_duration: DurationSec, max_ratio: u32) {
    assert_one_yocto();
    require!(self.is_owner_or_operators(), E002_NOT_ALLOWED);
    require!(self.data().state == RunningState::Running, E004_CONTRACT_PAUSED);

    let mut config = self.data().config.get().unwrap();
    // config.minimum_staking_duration_sec = min_duration;
    config.maximum_locking_duration_sec = max_duration;
    // config.min_booster_multiplier = min_ratio;
    config.max_locking_multiplier = max_ratio;

    config.assert_valid();
    self.data_mut().config.set(&config);
}
```

Listing 2.6: management.rs

```
pub fn stake_lock_seed(
    &mut self,
    farmer_id: &AccountId,
    seed_id: &SeedId,
    amount: u128,
    duration_sec: u32,
) {
    let mut farmer = self.internal_unwrap_farmer(&farmer_id);
    let mut seed = self.internal_unwrap_seed(&seed_id);
    require!(amount >= seed.min_deposit, E307_BELOW_MIN_DEPOSIT);


    require!(seed.min_locking_duration_sec > 0, E300_FORBID_LOCKING);
    require!(duration_sec >= seed.min_locking_duration_sec, E201_INVALID_DURATION);
    let config = self.internal_config();
    require!(duration_sec <= config.maximum_locking_duration_sec, E201_INVALID_DURATION);


    self.internal_do_farmer_claim(&mut farmer, &mut seed);


    let mut farmer_seed = farmer.seeds.get(&seed_id).unwrap();
    if farmer_seed.is_empty() {
        seed.farmer_count += 1;
    }
    let increased_seed_power = farmer_seed.add_lock(amount, duration_sec, &config);
    farmer.seeds.insert(&seed_id, &farmer_seed);


    seed.total_seed_amount += amount;
    seed.total_seed_power += increased_seed_power;


    self.update_impacted_seeds(&mut farmer, &seed_id);


    self.internal_set_farmer(&farmer_id, farmer);
    self.internal_set_seed(&seed_id, seed);


    Event::SeedDeposit {
        farmer_id,
        seed_id,
        deposit_amount: &U128(amount),
        increased_power: &U128(increased_seed_power),
        duration: duration_sec,
    }
```

```
        .emit();
}
```

Listing 2.7: token_receiver.rs

```
  #[payable]
pub fn lock_free_seed(&mut self, seed_id: SeedId, duration_sec: u32, amount: Option<U128>) {
    assert_one_yocto();
    require!(self.data().state == RunningState::Running, E004_CONTRACT_PAUSED);


    let farmer_id = env::predecessor_account_id();


    let mut farmer = self.internal_unwrap_farmer(&farmer_id);
    let mut seed = self.internal_unwrap_seed(&seed_id);


    require!(seed.min_locking_duration_sec > 0, E300_FORBID_LOCKING);
    require!(duration_sec >= seed.min_locking_duration_sec, E201_INVALID_DURATION);
    let config = self.internal_config();
    require!(duration_sec <= config.maximum_locking_duration_sec, E201_INVALID_DURATION);


    self.internal_do_farmer_claim(&mut farmer, &mut seed);


    let mut farmer_seed = farmer.seeds.get(&seed_id).unwrap();
    let amount = if let Some(request) = amount {
        request.0
    } else {
        farmer_seed.free_amount
    };


    let increased_seed_power =
        farmer_seed.free_to_lock(amount, duration_sec, &config);
    farmer.seeds.insert(&seed_id, &farmer_seed);


    seed.total_seed_power += increased_seed_power;


    self.update_impacted_seeds(&mut farmer, &seed_id);


    self.internal_set_farmer(&farmer_id, farmer);
    self.internal_set_seed(&seed_id, seed);


    Event::SeedFreeToLock {
        farmer_id: &farmer_id,
        seed_id: &seed_id,
        amount: &U128(amount),
        increased_power: &U128(increased_seed_power),
        duration: duration_sec,
    }
    .emit();
}
```

Listing 2.8: actions_of_farmer_seed.rs

Impact If `config.maximum_locking_duration_sec` is mistakenly configured to be less than the `min_locking_duration_sec` of certain `seed`s, users are not able to lock in those `seed`s.

Suggestion Add a check to ensure that `config.maximum_locking_duration_sec` is greater than the `min_locking_duration_sec` of all `seed`s when updating it.

Feedback If the updated `config.maximum_locking_duration_sec` is less than `seed.min_locking_duration_sec`, it can be considered a way to disable `seed` lock.

2.2 Additional Recommendation

2.2.1 Redundant Check in storage_withdraw()

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `strorage_withdraw()` is already disabled with panic. Thus, the checks `assert_one_yocto()` and `assert_contract_running()` are redundant.

```
    #[payable]
    fn storage_withdraw(
        &mut self,
        #[allow(unused_variables)] amount: Option<U128>,
    ) -> StorageBalance {
        assert_one_yocto();
        self.assert_contract_running();
        env::panic_str(E005_NOT_IMPLEMENTED);
    }
```

Listing 2.9: storage_impl.rs

Suggestion Remove the redundant check.

2.2.2 Lack of assert_one_yocto Check

Status Confirmed

Introduced by `Version 1`

Description The function `withdraw_seed()` includes the token transfer operation. Therefore, the check for `assert_one_yocto()` should be placed in the function `withdraw_seed()`.

```
     pub fn withdraw_seed(&mut self, seed_id: SeedId, amount: Option<U128>) -> Promise {
        require!(self.data().state == RunningState::Running, E004_CONTRACT_PAUSED);
        let farmer_id = env::predecessor_account_id();
```

```
    let mut farmer = self.internal_unwrap_farmer(&farmer_id);
    let withdraw_seed = farmer.withdraws.get(&seed_id).unwrap();
    let withdraw_amount: Balance = if let Some(amount) = amount {
        amount.into()
    } else {
        withdraw_seed.amount
    };
    farmer.sub_withdraw_seed(&seed_id, withdraw_amount, self.get_config().delay_withdraw_sec);
    self.internal_set_farmer(&farmer_id, farmer);
    self.transfer_seed_token(&farmer_id, &seed_id, withdraw_amount)
}
```

Listing 2.10: actions_of_farmer_seed.rs

Suggestion Add the check `assert_one_yocto()` to the function `withdraw_seed()`.

Feedback from the Project Assets in funciton `withdraw()` no longer generate earnings, so it doesn't need 1 `yocto`.

2.2.3 Lack of Minimum Deposit Check

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In function `unlock_and_unstake_seed()`, there is no check to verify if the remaining `free_amount` and `locked_amount` of the `farmer_seed` are still greater than `seed.min_deposit`. This can result in leftover dust, which is against the original design purpose.

```
 #[payable]
pub fn unlock_and_unstake_seed(
    &mut self,
    seed_id: SeedId,
    unlock_amount: U128,
    unstake_amount: U128,
) {
    assert_one_yocto();
    require!(self.data().state == RunningState::Running, E004_CONTRACT_PAUSED);


    let unlock_amount: Balance = unlock_amount.into();
    let unstake_amount: Balance = unstake_amount.into();


    let farmer_id = env::predecessor_account_id();


    let mut farmer = self.internal_unwrap_farmer(&farmer_id);
    let mut seed = self.internal_unwrap_seed(&seed_id);


    self.internal_do_farmer_claim(&mut farmer, &mut seed);


```

```
        let mut farmer_seed = farmer.seeds.get(&seed_id).unwrap();


        let prev = farmer_seed.get_seed_power();


        let decreased_seed_power =
        if unlock_amount > 0 {
            farmer_seed.unlock_to_free(unlock_amount)
        } else {
            0
        };
        if unstake_amount > 0 {
            farmer_seed.withdraw_free(unstake_amount);
            farmer.add_withdraw_seed(&seed_id, unstake_amount);
        }


        seed.total_seed_amount -= unstake_amount;
        seed.total_seed_power = seed.total_seed_power - prev + farmer_seed.get_seed_power();


        if farmer_seed.is_empty() {
            farmer.seeds.remove(&seed_id);
            if seed.farmer_count > 0 {
                seed.farmer_count -= 1;
            }
        } else {
            farmer.seeds.insert(&seed_id, &farmer_seed);
        }


        self.update_impacted_seeds(&mut farmer, &seed_id);


        self.internal_set_farmer(&farmer_id, farmer);
        self.internal_set_seed(&seed_id, seed);


        if unlock_amount > 0 {
            Event::SeedUnlock {
                farmer_id: &farmer_id,
                seed_id: &seed_id,
                unlock_amount: &U128(unlock_amount),
                decreased_power: &U128(decreased_seed_power),
                slashed_seed: &U128(0),
            }
            .emit();
        }
    }
```

Listing 2.11: actions_of_farmer_seed.rs

Suggestion Add checks to ensure that the amount of remaining funds is greater than `min_deposit` after unlocking/unstaking.