



# BlockSec

## Security Audit Report for Cornerstone

**Date:** September 21, 2022

**Version:** 1.0

**Contact:** [contact@blocksec.com](mailto:contact@blocksec.com)

# Contents

|          |   |          |
|----------|---|----------|
| <b>1</b> | <b>Introduction</b>   | <b>1</b> |
| 1.1      | About Target Contracts . . . . .  | 1        |
| 1.2      | Disclaimer . . . . .  | 4        |
| 1.3      | Procedure of Auditing . . . . .   | 4        |
| 1.3.1    | Software Security . . . . .   | 4        |
| 1.3.2    | DeFi Security . . . . .   | 5        |
| 1.3.3    | NFT Security . . . . .  | 5        |
| 1.3.4    | Additional Recommendation . . . . .   | 5        |
| 1.4      | Security Model . . . . .  | 5        |
| <b>2</b> | <b>Findings</b>   | <b>7</b> |
| 2.1      | Software Security . . . . .   | 7        |
| 2.1.1    | Improper Transfer Failure Handling during Unstaking . . . . .   | 7        |
| 2.1.2    | Potential DoS Problem . . . . .   | 8        |
| 2.1.3    | Lack of Callback Function for Function <code>mft_transfer_call()</code> . . . . .                         | 9        |
| 2.1.4    | Lack of Callback Function for Transferring NEAR . . . . .   | 10       |
| 2.2      | DeFi Security . . . . .   | 12       |
| 2.2.1    | Unchecked Token Address Used for Bidding . . . . .  | 12       |
| 2.2.2    | Unfair Reward Distribution . . . . .  | 14       |
| 2.2.3    | Incorrect Calculation of <code>tune()</code> in Inverse Bond . . . . .                                    | 15       |
| 2.2.4    | Duplicated Account Registration in Treasury . . . . .   | 17       |
| 2.2.5    | Missed Sanity Check in <code>bootstrap_liquidity()</code> . . . . .                                       | 18       |
| 2.2.6    | Inconsistency between Implementation and Documentation . . . . .  | 19       |
| 2.3      | Additional Recommendation . . . . .   | 19       |
| 2.3.1    | Potential Revert in Claiming . . . . .  | 19       |
| 2.3.2    | Unbalanced Gas Distribution in <code>internal_unstake()</code> . . . . .                                  | 20       |
| 2.3.3    | Missed Sanity Check in Auction . . . . .  | 22       |
| 2.3.4    | Incompatible Tokens . . . . .   | 23       |
| 2.3.5    | Timely <code>distribute()</code> upon the Epoch Change . . . . .  | 23       |
| 2.3.6    | Potential Centralization Problem . . . . .  | 25       |
| 2.3.7    | Missed Sanity Check in <code>set_stakeholder()</code> . . . . .   | 25       |
| 2.3.8    | Missed <code>assert_one_yocto()</code> in System Configuration . . . . .                                  | 28       |
| 2.4      | Notes . . . . .   | 30       |
| 2.4.1    | Delayed Price from Oracle . . . . .   | 30       |
| 2.4.2    | Inconsistency of Valuation between <code>corn_lp_token</code> and <code>general_lp_token</code> . . . . . | 31       |
| 2.4.3    | Unrestricted Staking Duration . . . . .   | 32       |

## Report Manifest

| Item   | Description     |
|--------|-----------------|
| Client | Cornerstone DAO |
| Target | Cornerstone     |

## Version History

| Version | Date               | Description   |
|---------|--------------------|---------------|
| 1.0     | September 21, 2022 | First Release |

**About BlockSec** The **BlockSec Team** focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team 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 released detailed analysis reports of high-impact security incidents. 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 repository that has been audited includes `cornerstone` <sup>1</sup>.

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  |
|-------------|------------------------|---|
| Cornerstone | <code>Version 1</code> | <code>406eeee879713af199e7877de0880b057ba0a09c</code> |
|             | <code>Version 2</code> | <code>4a5d7f0544a224eed3f02a49858bd6a098d858b0</code> |
|             | <code>Version 3</code> | <code>9c43f8aebd3b8a1bc708c125e341f2899d7b3cc4</code> |

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

```
1 contracts
2 |-- allocators
3 | |-- burrow-allocator
4 | | |-- src
5 | |     |-- allocate.rs
6 | |     |-- config.rs
7 | |     |-- interfaces
8 | |     | |-- burrow.rs
9 | |     | |-- ft.rs
10 | |     | |-- mod.rs
11 | |     | |-- treasury.rs
12 | |     |-- lib.rs
13 |-- linear-allocator
14 | | |-- src
15 | |     |-- allocate.rs
16 | |     |-- config.rs
17 | |     |-- interfaces
18 | |     | |-- ft.rs
19 | |     | |-- linear.rs
20 | |     | |-- mod.rs
21 | |     | |-- treasury.rs
22 | |     |-- lib.rs
23 |-- ref-allocator
24 | | |-- src
```

---

<sup>1</sup><https://github.com/corndao/corndao>

```
25 | | | |-- allocate.rs
26 | | | |-- config.rs
27 | | | |-- events.rs
28 | | | |-- gas.rs
29 | | | |-- interfaces
30 | | | | |-- ft.rs
31 | | | | |-- mft.rs
32 | | | | |-- mod.rs
33 | | | | |-- ref_finance.rs
34 | | | | |-- treasury.rs
35 | | | |-- lib.rs
36 | | | |-- token_receiver.rs
37 | |-- wnear-allocator
38 | | |-- src
39 | | | |-- allocate.rs
40 | | | |-- config.rs
41 | | | |-- interfaces
42 | | | | |-- ft.rs
43 | | | | |-- mod.rs
44 | | | | |-- treasury.rs
45 | | | | |-- wnear.rs
46 | | | |-- lib.rs
47 | |-- xcorn-allocator (folder added in version 2)
48 | | |-- src
49 | | | |-- allocate.rs
50 | | | |-- config.rs
51 | | | |-- interfaces
52 | | | | |-- ft.rs
53 | | | | |-- mod.rs
54 | | | | |-- treasury.rs
55 | | | | |-- xcorn.rs
56 | | | |-- lib.rs
57 |-- common
58 | |-- src
59 | | |-- balance_tracker.rs
60 | | |-- epoch.rs
61 | | |-- lib.rs
62 | | |-- number.rs
63 | | |-- timestamp.rs
64 | | |-- token.rs
65 |-- corn
66 | |-- src
67 | | |-- active_vector.rs
68 | | |-- auction.rs
69 | | |-- big_decimal.rs
70 | | |-- bonding
71 | | | |-- bond.rs
72 | | | |-- bond_note.rs
73 | | | |-- inverse_bond.rs
74 | | | |-- market.rs
75 | | | |-- mod.rs
76 | | | |-- redeem.rs
77 | | | |-- valuation.rs
```

```
78 |      |-- events.rs
79 |      |-- fungible_token
80 |      | |-- core.rs
81 |      | |-- metadata.rs
82 |      | |-- mod.rs
83 |      | |-- storage.rs
84 |      |-- interfaces
85 |      | |-- ft.rs
86 |      | |-- mft.rs
87 |      | |-- mod.rs
88 |      | |-- oracle.rs
89 |      | |-- vecorn.rs
90 |      | |-- xcorn.rs (file added in version 2)
91 |      |-- lib.rs
92 |      |-- manage.rs
93 |      |-- token_receiver.rs
94 |      |-- treasury.rs
95 |      |-- types.rs
96 |      |-- upgrade.rs
97 |      |-- utils.rs
98 |-- vecorn
99 | |-- src
100 |     |-- account.rs
101 |     |-- config.rs
102 |     |-- events.rs
103 |     |-- fungible_token
104 |     | |-- core.rs
105 |     | |-- metadata.rs
106 |     | |-- mod.rs
107 |     |-- history.rs
108 |     |-- interfaces
109 |     | |-- ft.rs
110 |     | |-- mod.rs
111 |     |-- lib.rs
112 |     |-- lock.rs
113 |     |-- rewards.rs
114 |     |-- token_receiver.rs
115 |     |-- types.rs
116 |     |-- upgrade.rs
117 |     |-- utils.rs
118 |-- xcorn
119 | |-- src
120 |     |-- config.rs
121 |     |-- distributor.rs
122 |     |-- events.rs
123 |     |-- fungible_token
124 |     | |-- core.rs
125 |     | |-- metadata.rs
126 |     | |-- mod.rs
127 |     | |-- price.rs
128 |     | |-- storage.rs
129 |     |-- interfaces
130 |     |-- corn.rs
```

```
131 | |-- ft.rs
132 | |-- mod.rs
133 | |-- lib.rs
134 | |-- policy.rs
135 | |-- stake.rs
136 | |-- token_receiver.rs
137 | |-- types.rs
138 | |-- upgrade.rs
139 | |-- utils.rs
```

**Listing 1.1:** Audit Scope for this Report

## 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 <sup>2</sup> and Common Weakness Enumeration <sup>3</sup>. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to

---

<sup>2</sup>[https://owasp.org/www-community/OWASP\\_Risk\\_Rating\\_Methodology](https://owasp.org/www-community/OWASP_Risk_Rating_Methodology)

<sup>3</sup><https://cwe.mitre.org/>



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.

## Chapter 2 Findings

In total, we find **ten** potential issues. We also have **eight** recommendations and **three** note as follows:

- High Risk: 2
- Medium Risk: 5
- Low Risk: 3
- Recommendations: 8
- Notes: 3

| ID | Severity | Description   | Category          | Status    |
|----|----------|---|-------------------|-----------|
| 1  | Low      | Improper Transfer Failure Handling during Unstaking   | Software Security | Fixed     |
| 2  | Medium   | Potential DoS Problem   | Software Security | Fixed     |
| 3  | Medium   | Lack of Callback Function for Function <code>mft_transfer_call()</code>                         | Software Security | Fixed     |
| 4  | Medium   | Lack of Callback Function for Transferring NEAR   | Software Security | Fixed     |
| 5  | High     | Unchecked Token Address Used for Bidding  | DeFi Security     | Fixed     |
| 6  | Low      | Unfair Reward Distribution  | DeFi Security     | Fixed     |
| 7  | Low      | Incorrect Calculation of <code>tune()</code> in Inverse Bond                                    | DeFi Security     | Fixed     |
| 8  | High     | Duplicated Account Registration in Treasury   | DeFi Security     | Fixed     |
| 9  | Medium   | Missed Sanity Check in <code>bootstrap_liquidity()</code>                                       | DeFi Security     | Fixed     |
| 10 | Medium   | Inconsistency between Implementation and Documentation  | DeFi Security     | Fixed     |
| 11 | -        | Potential Revert in Claiming  | Recommendation    | Fixed     |
| 12 | -        | Unbalanced Gas Distribution in <code>internal_unstake()</code>                                  | Recommendation    | Fixed     |
| 13 | -        | Missed Sanity Check in Auction  | Recommendation    | Fixed     |
| 14 | -        | Incompatible Tokens   | Recommendation    | Confirmed |
| 15 | -        | Timely <code>distribute()</code> upon the Epoch Change  | Recommendation    | Confirmed |
| 16 | -        | Potential Centralization Problem  | Recommendation    | Confirmed |
| 17 | -        | Missed Sanity Check in <code>set_stakeholder()</code>   | Recommendation    | Fixed     |
| 18 | -        | Missed <code>assert_one_yocto()</code> in System Configuration                                  | Recommendation    | Fixed     |
| 19 | -        | Delayed Price from Oracle   | Note              | Confirmed |
| 20 | -        | Inconsistency of Valuation between <code>corn_lp_token</code> and <code>general_lp_token</code> | Note              | Confirmed |
| 21 | -        | Unrestricted Staking Duration   | Note              | Confirmed |

The details are provided in the following sections.

### 2.1 Software Security

#### 2.1.1 Improper Transfer Failure Handling during Unstaking

**Severity** Low

**Status** Fixed in [Version 2](#)

Introduced by [Version 1](#)

**Description** Function `on_unstake()` will recover the amount of `total_staked_corn` and the user's `num_shares` when the cross-contract invocation `ft_transfer()` executed in block `N` is failed. However, function `on_unstake()` is executed in block `N+1`. The exchange ratio between `Corn` token and `Xcorn` token may be different between block `N+1` and block `N`. Therefore, it's unreasonable to mint the outdated `num_shares` back to the user.

```
73  #[private]
74  pub fn on_unstake(&mut self, account_id: AccountId, receive_amount: U128, num_shares: U128) {
75      let receive_amount = receive_amount.into();
76      let num_shares = num_shares.into();
77
78      if is_promise_success() {
79          // Decrease total staked amount at current epoch
80          self.staked_corn_tracker
81              .decrease_current_epoch_balance(self.internal_distributor().epoch, receive_amount)
82              ;
83
84          // Emit event only if ft_transfer succeeds
85          Event::Unstake {
86              account_id: &account_id,
87              unstaked_amount: &U128(receive_amount),
88              burnt_stake_shares: &U128(num_shares),
89              new_stake_shares: &U128(self.internal_ft_balance(&account_id)),
90          }
91          .emit();
92          log!(
93              "Contract total staked balance is {}. Total number of shares {}",
94              self.total_staked_corn,
95              self.xcorn_total_supply()
96          );
97      } else {
98          // If transfer failed, undo unstake:
99          // 1. rollback the $xCORN burn operation
100         // 2. rollback the decreased total staked $CORN amount
101         self.mint_xcorn(&account_id, num_shares, Some("undo unstake"));
102         self.total_staked_corn += receive_amount;
103     }
```

**Listing 2.1:** contracts/xcorn/src/stake.rs

**Impact** Users may get incorrect shares in function `on_unstake()`.

**Suggestion I** Re-calculate the shares in function `on_unstake()`.

## 2.1.2 Potential DoS Problem

**Severity** Medium

**Status** Fixed in [Version 2](#)

Introduced by [Version 1](#)

**Description** In the process of bonding, if the user purchases a certain amount of [Corn](#) tokens, the contract will not mint or transfer the [Corn](#) tokens to the user directly. Instead, it will record the purchased amount in the contract state (i.e., [BondNote](#)) and the user will claim the purchased [Corn](#) tokens later. The record will increase the storage usage of the contract.

However, there is no limit for the minimum purchase amount and the purchase times. In this case, malicious users can use up the storage of contract by purchasing the bonds with a small amount of [quote\\_token](#) (e.g., 1 yocto) repeatedly.

```
87 pub fn insert(  
88     &mut self,  
89     account_id: &AccountId,  
90     market_id: MarketId,  
91     note: &mut BondNote,  
92 ) -> u64 {  
93     let key = self.lookup_key(account_id, market_id);  
94     let mut user_notes = self.notes.get(&key).unwrap_or_else(|| {  
95         ActiveVector::new(StorageKey::UserNotes {  
96             account_id: account_id.clone(),  
97             market_id,  
98         })  
99     });  
100  
101     let index = user_notes.items_count();  
102     note.id = index;  
103     user_notes.append(note);  
104     self.notes.insert(&key, &user_notes);  
105  
106     index  
107 }
```

**Listing 2.2:** contracts/corn/src/bonding/bond\_note.rs

**Impact** The [Corn](#) token contract may run out of the storage, resulting in DoS.

**Suggestion I** It's suggested to restrict the minimum purchase amount in the bonding.

### 2.1.3 Lack of Callback Function for Function [mft\\_transfer\\_call\(\)](#)

**Severity** Medium

**Status** Fixed in [Version 2](#)

**Introduced by** [Version 1](#)

**Description** In contract [ref-allocator](#), the function [deposit\\_lpt\\_to\\_farm\(\)](#) invokes a cross-contract call (i.e., function [mft\\_transfer\\_call\(\)](#)) to the [ref\\_exchange\\_contract](#). However, it doesn't implement the callback function [on\\_deposited\\_to\\_farm\(\)](#) to handle the result.

```
224 pub fn deposit_lpt_to_farm(&mut self, token_id: String, amount: U128) -> Promise {  
225     self.assert_policy_team();  
226     require!(env::prepaid_gas() >= GAS_DEPOSIT_TO_FARM, ERR_NO_ENOUGH_GAS);  
227  
228     require!(  
229         self.get_lpt_liquid_amount(&token_id) >= amount.into(),
```

```
230         ERR_NO_ENOUGH_LPT_BALANCE
231     );
232
233     self.decrease_lpt_liquid_amount(&token_id, amount.into());
234
235     ref_mft::ext(self.get_config().ref_exchange_contract_id)
236         .with_unused_gas_weight(8)
237         .mft_transfer_call(
238             token_id,
239             self.get_config().ref_farming_contract_id,
240             amount,
241             None,
242             "\"Free\"".to_string(),
243         )
244     }
245
246     #[private]
247     pub fn on_deposited_to_farm(&mut self, token_id: String, amount: U128) {
248         if is_promise_success() {
249             self.increase_lpt_in_farm_amount(&token_id, amount.into());
250
251             Event::LPTDepositedToFarm {
252                 token_id: &token_id,
253                 amount: &amount,
254             }
255             .emit();
256         } else {
257             self.increase_lpt_liquid_amount(&token_id, amount.into());
258         }
259     }
```

**Listing 2.3:** contracts/allocators/ref-allocator/src/allocate.rs

**Impact** The contract state will not be updated when the cross-contract invocation succeeds or fails.

**Suggestion I** Implement the callback function.

## 2.1.4 Lack of Callback Function for Transferring NEAR

**Severity** Medium

**Status** Fixed in [Version 2](#)

**Introduced by** [Version 1](#)

**Description** In function `manage_asset()`, allocators are allowed to transfer authorized assets out of the treasury. However, it doesn't implement the callback function when the asset is [NEAR](#) (lines 236 - 245).

```
224     pub fn manage_asset(&mut self, token: Token, amount: U128) -> PromiseOrValue<U128> {
225         let manager_id = env::predecessor_account_id();
226         let managable_tokens = self
227             .permissions
228             .get(&manager_id.clone())
229             .expect(ERR_NOT_ALLOWED_TO_MANAGE);
230         require!(managable_tokens.contains(&token), ERR_NOT_ALLOWED_TO_MANAGE);
```

```
231
232     let amount: Balance = amount.into();
233     require!(amount > 0, ERR_BAD_MANAGED_AMOUNT);
234
235     match token {
236         Token::NEAR => {
237             // NEAR native token
238             Promise::new(manager_id.clone()).transfer(amount);
239             Event::ManageAsset {
240                 manager_id: &manager_id,
241                 token: &token,
242                 amount: &U128(amount),
243             }
244             .emit();
245             PromiseOrValue::Value(U128(amount))
246         }
247         Token::NEP141 { ref token_address } => {
248             // NEP141 token can be $CORN
249             if token_address.clone() == env::current_account_id() {
250                 self.tokens.internal_register_account(&manager_id);
251                 self.tokens.internal_transfer(
252                     &env::current_account_id(),
253                     &manager_id,
254                     amount,
255                     Some("manage".to_string()),
256                 );
257                 Event::ManageAsset {
258                     manager_id: &manager_id,
259                     token: &token,
260                     amount: &U128(amount),
261                 }
262                 .emit();
263                 PromiseOrValue::Value(U128(amount))
264             } else {
265                 nep141::ext(token_address.clone())
266                     .with_attached_deposit(ONE_YOCTO)
267                     .with_unused_gas_weight(4)
268                     .ft_transfer(
269                         manager_id.clone(),
270                         amount.into(),
271                         Some("manage".to_string()),
272                     )
273                     .then(
274                         Self::ext(env::current_account_id())
275                             .with_unused_gas_weight(1)
276                             .on_manage_asset(manager_id.clone(), token, amount.into()),
277                     )
278                     .into()
279             }
280         }
281         Token::RefMFT {
282             ref token_address,
283             ref token_id,
```

```
284         } => ref_mft::ext(token_address.clone())
285             .with_attached_deposit(ONE_YOCTO)
286             .with_unused_gas_weight(4)
287             .mft_transfer(
288                 token_id.clone(),
289                 manager_id.clone(),
290                 amount.into(),
291                 Some("manage".to_string()),
292             )
293             .then(
294                 Self::ext(env::current_account_id())
295                     .with_unused_gas_weight(1)
296                     .on_manage_asset(manager_id.clone(), token, amount.into()),
297             )
298             .into(),
299     }
300 }
```

**Listing 2.4:** contracts/corn/src/treasury.rs

**Impact** If the transfer fails, allocators would not know, and keep executing the planned operations with its own [NEAR](#) tokens.

**Suggestion I** Implement the callback function.

## 2.2 DeFi Security

### 2.2.1 Unchecked Token Address Used for Bidding

**Severity** High

**Status** Fixed in [Version 2](#)

**Introduced by** [Version 1](#)

**Description** In each auction, the [token\\_address](#) of the required [quote\\_token](#), which is used for bidding, is fixed and set during the creation. However, the contract does not check whether the token transferred in by users is the [quote\\_token](#). In this case, attackers can bid for [Corn](#) with any tokens unrestrictedly.

```
66     #[near_bindgen]
67 impl FungibleTokenReceiver for Corn {
68     fn ft_on_transfer(
69         &mut self,
70         sender_id: AccountId,
71         amount: U128,
72         msg: String,
73     ) -> PromiseOrValue<U128> {
74         if msg.is_empty() {
75             // refund all
76             return PromiseOrValue::Value(amount);
77         }
78
79         let token_address = env::predecessor_account_id();
80         let quote_token = Token::NEP141 { token_address };
```

```
81     let message = serde_json::from_str::<OnTransferAction>(&msg).expect(ERR_MALFORMED_MESSAGE);
82
83     match message {
84         OnTransferAction::Bond {
85             market_id,
86             max_price,
87         } => self.on_purchase_request(
88             &sender_id,
89             market_id,
90             quote_token,
91             amount.into(),
92             max_price.into(),
93         ),
94         OnTransferAction::Bid { auction_id } => {
95             self.on_bid_request(&sender_id, auction_id, amount.into())
96         }
97     }
98 }
99 }
```

**Listing 2.5:** contracts/corn/src/token\_receiver.rs

```
483 pub(crate) fn on_bid_request(
484     &mut self,
485     user_id: &AccountId,
486     auction_id: AuctionId,
487     amount: Balance,
488 ) -> PromiseOrValue<U128> {
489     let mut auction = self.get_auction(auction_id);
490     let result = auction.bid(user_id, amount);
491     if let Err(err) = result {
492         panic!("{}", err);
493     }
494     self.auctions.replace(auction_id, &auction); // re-write auction
495
496     Event::AuctionBid {
497         id: auction_id,
498         account_id: user_id,
499         amount: &U128(amount),
500     }
501     .emit();
502
503     PromiseOrValue::Value(U128(0))
504 }
```

**Listing 2.6:** contracts/corn/src/auction.rs

**Impact** **Corn** can be purchased by malicious users with worthless tokens in the auction.

**Suggestion I** Check whether the token transferred in by users matches the requirement of the auction market, if not, abort the transaction.



## 2.2.2 Unfair Reward Distribution

**Severity** Low

**Status** Fixed in [Version 3](#)

**Introduced by** [Version 1](#)

**Description** In function `internal_stake()`, the shares of the user are calculated and recorded before the reward distribution. If the user stakes to trigger the reward distribution, the calculation of the reward amount will not include the user's newly staked [Corn](#). However, the user can share the reward with others, which is unfair.

```
108 pub(crate) fn internal_stake(&mut self, account_id: &AccountId, amount: Balance) {
109     let min_gas = GAS_FOR_STAKE + GAS_FOR_DISTRIBUTE;
110     require!(
111         env::prepaid_gas() >= min_gas,
112         format!("{}", require at least {:?})", ERR_NO_ENOUGH_GAS, min_gas)
113     );
114
115     require!(amount > 0, ERR_NON_POSITIVE_STAKING_AMOUNT);
116
117     // Calculate the number of "stake" shares that the account will receive for staking the
118     // given amount.
119     let num_shares = self.amount_to_shares(amount, false);
120     require!(num_shares > 0, ERR_NON_POSITIVE_CALCULATED_STAKING_SHARE);
121
122     // Distribute rewards if the method is called at the very beginning of the epoch
123     self.internal_distribute();
124
125     // Mint $xCORN for the account
126     self.mint_xcorn(account_id, num_shares, Some("stake"));
127
128     // Increase total staked $CORN
129     self.total_staked_corn += amount;
130     // Increase total staked amount at current epoch
131     self.staked_corn_tracker
132         .increase_current_epoch_balance(self.internal_distributor().epoch, amount);
133
134     Event::Stake {
135         account_id,
136         staked_amount: &U128(amount),
137         minted_stake_shares: &U128(num_shares),
138         new_stake_shares: &U128(self.internal_ft_balance(account_id)),
139     }
140     .emit();
141     log!(
142         "Contract total staked balance is {}. Total number of shares {}",
143         self.total_staked_corn,
144         self.xcorn_total_supply()
145     );
146 }
```

**Listing 2.7:** contracts/xcorn/src/stake.rs

**Impact** Stakers have to share their rewards with the user who stakes to trigger the reward distribution.

**Suggestion I** Include the user's newly staked [Corn](#) when calculating the reward.

**Suggestion II** Distribute the rewards before the user stakes to trigger the distribution.

### 2.2.3 Incorrect Calculation of `tune()` in Inverse Bond

**Severity** Low

**Status** Fixed in [Version 2](#)

**Introduced by** [Version 1](#)

**Description** In the market of `inverse_bond`, when a user purchases the bond to trigger the function `tune()`, the parameter `(payout_total_supply + payout)` will be input as a new `total_supply` of `payout` token to calculate a new `bcv`. However, different from the `bond` market, the `payout` token in the `inverse_bond` market will be transferred to purchasers instead of minting. That's to say, the `total_supply` of the `payout` token will not increase in the `inverse_bond` market, which is against the implementation.

```
218 fn internal_purchase_inverse_bond(  
219     &mut self,  
220     account_id: &AccountId,  
221     inverse_market_id: MarketId,  
222     quote_token_amount: Balance, // CORN amount  
223     max_price: Balance,  
224 ) -> Balance {  
225     let mut inverse_market = self.internal_get_inverse_market(inverse_market_id);  
226  
227     // do purchase  
228     let payout_total_supply = self  
229         .inverse_market_payout_supply  
230         .get(&inverse_market_id)  
231         .expect(ERR_INVERSE_MARKET_MISS_PAYOUT_SUPPLY);  
232     let payout = inverse_market.purchase(quote_token_amount, max_price, payout_total_supply);  
233  
234     // save inverse market  
235     self.inverse_markets  
236         .update(inverse_market_id, &inverse_market);  
237  
238     // save payouts  
239     self.internal_inc_inverse_market_payout(account_id, inverse_market_id, payout);  
240  
241     Event::InverseBondPurchased {  
242         account_id,  
243         inverse_market_id,  
244         amount: &U128(quote_token_amount),  
245         payout: &U128(payout),  
246     }  
247     .emit();  
248  
249     // if current purchase closed the market  
250     if inverse_market.capacity == 0 {  
251         Event::InverseMarketClosed {  
252             inverse_market_id,
```

```
253         total_debt: &U128(inverse_market.total_debt),
254         max_debt: &U128(inverse_market.max_debt),
255     }
256     .emit();
257 }
258
259 payout
260 }
```

**Listing 2.8:** corn/src/bonding/inverse\_bond.rs

```
205 pub fn purchase(
206     &mut self,
207     quote_token_amount: Balance,
208     max_price: Balance,
209     payout_total_supply: Balance,
210 ) -> Balance {
211     require!(
212         self.conclusion > current_timestamp_ms(),
213         ERR_MARKET_CONCLUDED
214     );
215     require!(self.capacity > 0, ERR_MARKET_CLOSED);
216
217     // decay market debt and BCV over time
218     self.decay();
219
220     let price = self
221         .price(current_timestamp_ms(), payout_total_supply)
222         .round_u128();
223     require!(price <= max_price, ERR_PRICE_SLIPPAGE);
224
225     // payout CORN token amount
226     let payout = self.calculate_payout(quote_token_amount, price);
227     require!(payout <= self.max_payout, ERR_EXCEED_MAX_PAYOUT);
228     require!(payout <= self.capacity, ERR_EXCEED_CAPACITY);
229
230     self.capacity -= payout;
231
232     self.purchased += quote_token_amount;
233     self.sold += payout;
234
235     self.total_debt += payout;
236
237     if self.total_debt > self.max_debt {
238         self.capacity = 0;
239     } else {
240         // mint hasn't happened yet, we need to manually add payout to total supply when tune
241         self.tune(payout_total_supply + payout);
242     }
243
244     payout
245 }
```

**Listing 2.9:** corn/src/bonding/market.rs

**Impact** The implementation is against the reality as the total supply of the sale tokens would not be affected by the bonding market.

**Suggestion I** The value of the input parameter in function `tune()` should stay unchanged.

## 2.2.4 Duplicated Account Registration in Treasury

**Severity** High

**Status** Fixed in [Version 2](#)

**Introduced by** [Version 1](#)

**Description** In function `manage_asset()`, authorized allocators are allowed to transfer `Corn` tokens out for the investing purpose. However, each time the allocator tries to transfer the `Corn` tokens out, the `Corn` token contract will invoke the function `internal_register_account()` to register an account for the allocator even if the allocator already has an account. As a result, function `internal_register_account()` will panic, and the allocator who already has an account cannot withdraw `Corn` tokens.

```
224 pub fn manage_asset(&mut self, token: Token, amount: U128) -> PromiseOrValue<U128> {
225     let manager_id = env::predecessor_account_id();
226     let managable_tokens = self
227         .permissions
228         .get(&manager_id.clone())
229         .expect(ERR_NOT_ALLOWED_TO_MANAGE);
230     require!(managable_tokens.contains(&token), ERR_NOT_ALLOWED_TO_MANAGE);
231
232     let amount: Balance = amount.into();
233     require!(amount > 0, ERR_BAD_MANAGED_AMOUNT);
234
235     match token {
236         Token::NEAR => {
237             // NEAR native token
238             Promise::new(manager_id.clone()).transfer(amount);
239             Event::ManageAsset {
240                 manager_id: &manager_id,
241                 token: &token,
242                 amount: &U128(amount),
243             }
244             .emit();
245             PromiseOrValue::Value(U128(amount))
246         }
247         Token::NEP141 { ref token_address } => {
248             // NEP141 token can be $CORN
249             if token_address.clone() == env::current_account_id() {
250                 self.tokens.internal_register_account(&manager_id);
251                 self.tokens.internal_transfer(
252                     &env::current_account_id(),
253                     &manager_id,
254                     amount,
255                     Some("manage".to_string()),
256                 );
257                 Event::ManageAsset {
258                     manager_id: &manager_id,
```

```
259         token: &token,
260         amount: &U128(amount),
261     }
262     .emit();
263     PromiseOrValue::Value(U128(amount))
264 } else {
265     nep141::ext(token_address.clone())
266     .with_attached_deposit(ONE_YOCTO)
267     .with_unused_gas_weight(4)
268     .ft_transfer(
269         manager_id.clone(),
270         amount.into(),
271         Some("manage".to_string()),
272     )
273     .then(
274         Self::ext(env::current_account_id())
275         .with_unused_gas_weight(1)
276         .on_manage_asset(manager_id.clone(), token, amount.into()),
277     )
278     .into()
279 }
280 }
281 Token::RefMFT {
282     ref token_address,
283     ref token_id,
284 } => ref_mft::ext(token_address.clone())
285     .with_attached_deposit(ONE_YOCTO)
286     .with_unused_gas_weight(4)
287     .mft_transfer(
288         token_id.clone(),
289         manager_id.clone(),
290         amount.into(),
291         Some("manage".to_string()),
292     )
293     .then(
294         Self::ext(env::current_account_id())
295         .with_unused_gas_weight(1)
296         .on_manage_asset(manager_id.clone(), token, amount.into()),
297     )
298     .into(),
299 }
300 }
```

**Listing 2.10:** corn/src/treasury.rs

**Impact** The authorized allocators can only withdraw [Corn](#) tokens once at most.

**Suggestion I** If the allocator already has an account, transfer requested [Corn](#) tokens directly.

## 2.2.5 Missed Sanity Check in `bootstrap_liquidity()`

**Severity** Medium

**Status** Fixed in [Version 2](#)

**Introduced by** Version 1

**Description** According to the design, the `total_supply` of `Corn` tokens is capped with `total_mintable`, and `total_mintable` can only be initialized once in function `bootstrap_liquidity()`. However, auctions will be held earlier than `bootstrap_liquidity()`, and they will mint `Corn` tokens for bidders, which increases the `total_supply`.

```
67 #[payable]
68 pub fn bootstrap_liquidity(&mut self, amount: U128, total_mintable: U128) {
69     assert_one_yocto();
70     self.assert_policy_team();
71
72     require!(self.total_mintable == 0, ERR_ALREADY_BOOTSTRAPPED);
73     require!(
74         amount.0 <= MAX_BOOTSTRAP_MINT_CORN * ONE_CORN,
75         ERR_BAD_BOOTSTRAP_MINT_AMOUNT
76     );
77     require!(total_mintable.0 > amount.0, ERR_BAD_TOTAL_MINTABLE);
78
79     self.total_mintable = total_mintable.into();
80     // Minted CORN in treasury will be used to create liquidity pool via allocator
81     self.mint_corn(&env::current_account_id(), amount.into(), Some("bootstrap"));
82 }
```

**Listing 2.11:** corn/src/treasury.rs

**Impact** The reward distribution of the whole system would not work.

**Suggestion I** Modify the requirement (line 77) to make sure `total_supply + amount.0 <= total_mintable`.

## 2.2.6 Inconsistency between Implementation and Documentation

**Severity** Medium

**Status** Fixed in Version 2

**Introduced by** Version 1

**Description** According to the documentation, users can burn `Corn` tokens for assets in the `treasury` anytime. However, according to the current implementation, users can only trade `Corn` for assets when the market of `inverse_bond` is opened, and the traded `Corn` tokens will not be burnt.

**Suggestion I** Implement corresponding features mentioned in the documentation.

## 2.3 Additional Recommendation

### 2.3.1 Potential Revert in Claiming

**Status** Fixed in Version 2

**Introduced by** Version 1

**Description** In the auction, users can claim their purchased `Corn` tokens via the function `auction_claim_corn()`. This function will check all auctions that are sold out to transfer claimable `Corn` tokens to users altogether.

However, users are not allowed to claim before `claimable_timestamp`, and each auction might have different `claimable_timestamp`, so even if some of the auctions are claimable, users still have to wait, or the invoking will be reverted.

```
387 pub fn auction_claim_corn(&mut self) {
388     let account_id = &env::predecessor_account_id();
389
390     let mut total_amount = 0;
391     for auction in self.auctions.to_vec().iter_mut() {
392         if auction.is_soldout() {
393             total_amount += auction.user_claim_corn(account_id);
394             self.auctions.replace(auction.id, auction);
395         }
396     }
397
398     require!(total_amount > 0, ERR_NO_CLAIMABLE);
399
400     // register user account if not
401     if !self.tokens.accounts.contains_key(account_id) {
402         self.tokens.internal_register_account(account_id);
403     }
404
405     // transfer CORN to user
406     self.tokens.internal_transfer(
407         &env::current_account_id(),
408         account_id,
409         total_amount,
410         Some("Auction claim".to_string()),
411     );
412
413     Event::AuctionCornClaimed {
414         account_id,
415         amount: &U128(total_amount),
416     }
417     .emit();
418 }
```

**Listing 2.12:** contracts/corn/src/auction.rs

**Suggestion I** It's suggested to skip the auction that is unready to claim, and distribute claimable `Corn` tokens to users first.

### 2.3.2 Unbalanced Gas Distribution in `internal_unstake()`

**Status** Fixed in [Version 2](#)

**Introduced by** [Version 1](#)

**Description** In function `internal_unstake()`, the gas for invoking cross-contract call `ft_transfer()` and callback function are specified as  $8 * 10^{12}$  and  $10 * 10^{12}$  respectively.

```
26/// Amount of gas for fungible token transfers
27pub const GAS_FOR_FT_TRANSFER: Gas = Gas(8 * Gas::ONE_TERA.0);
28/// Amount of gas for stake
```

```
29 pub const GAS_FOR_STAKE: Gas = Gas(6 * Gas::ONE_TERA.0);
30 /// Amount of gas for unstake and callback
31 pub const GAS_FOR_UNSTAKE: Gas = Gas(8 * Gas::ONE_TERA.0);
32 pub const GAS_FOR_ON_UNSTAKE: Gas = Gas(10 * Gas::ONE_TERA.0);
```

**Listing 2.13:** contracts/xcorn/src/stake.rs

However, in the implementation, a weight of 1 is specified for function `ft_transfer()`, and a weight of 2 is specified for function `on_unstake()`, which is disproportionate compared with the constants set before.

```
149 pub(crate) fn internal_unstake(&mut self, account_id: &AccountId, num_shares: Shares) {
150     // Ensure enough prepaid gas to transfer $CORN back to account
151     let min_gas =
152         GAS_FOR_FT_TRANSFER + GAS_FOR_UNSTAKE + GAS_FOR_ON_UNSTAKE + GAS_FOR_DISTRIBUTE;
153     require!(
154         env::prepaid_gas() >= min_gas,
155         format!("{}", require_at_least {:?}", ERR_NO_ENOUGH_GAS, min_gas)
156     );
157
158     require!(self.total_staked_corn > 0, ERR_CONTRACT_NO_STAKED_BALANCE);
159     require!(num_shares > 0, ERR_NON_POSITIVE_CALCULATED_UNSTAKING_SHARE);
160     require!(
161         self.internal_ft_balance(account_id) >= num_shares,
162         ERR_NO_ENOUGH_STAKED_BALANCE
163     );
164
165     // Calculating the amount of tokens the account will receive by unstaking the corresponding
166     // number of "stake" shares, rounding down.
167     let receive_amount = self.shares_to_amount(num_shares, false);
168     require!(
169         receive_amount > 0,
170         ERR_NON_POSITIVE_CALCULATED_STAKED_AMOUNT
171     );
172
173     // Distribute rewards if the method is called at the very beginning of the epoch.
174     // If's OK to distribute rewards even if ft_transfer() below fails, so we don't call the
175     // method
176     // in on_unstake()
177     self.internal_distribute();
178
179     // Burn stake shares ($xCORN)
180     self.burn_xcorn(account_id, num_shares, Some("unstake"));
181
182     // Decrease total staked $CORN amount
183     self.total_staked_corn -= receive_amount;
184
185     // Transfer unstaked $CORN to the account
186     // The account data update will be done in the callback
187     ext_fungible_token::ext(self.internal_config().corn_account_id)
188         .with_attached_deposit(1)
189         .with_unused_gas_weight(1)
190         .ft_transfer(
191             account_id.clone(),
192             U128(receive_amount),
```



```
192         Some("unstake".to_string()),
193     )
194     .then(
195         Self::ext(env::current_account_id())
196             .with_unused_gas_weight(2)
197             .on_unstake(account_id.clone(), U128(receive_amount), U128(num_shares)),
198     );
199 }
```

**Listing 2.14:** contracts/xcorn/src/stake.rs

**Suggestion I** Assign the gas weight accordingly.

### 2.3.3 Missed Sanity Check in Auction

**Status** Fixed in [Version 2](#)

**Introduced by** [Version 1](#)

**Description** Function `update_auction()` is a privileged function that allows the `policy team` to adjust the configuration of the auction before it starts. However, it doesn't check the adjusted `start_timestamp` after the modification.

```
316 pub fn update_auction(
317     &mut self,
318     auction_id: AuctionId,
319     supply_offered: Option<U128>,
320     start_timestamp: Option<Timestamp>,
321     end_timestamp: Option<Timestamp>,
322     start_price: Option<U128>,
323     end_price: Option<U128>,
324 ) -> Auction {
325     self.assert_policy_team();
326
327     let mut auction = self.get_auction(auction_id);
328     require!(
329         current_timestamp_ms() < auction.start_timestamp,
330         ERR_CANNOT_UPDATE_AUCTION_NOW
331     );
332
333     // setting supply offered to 0 means to disable this auction
334     if let Some(supply_offered) = supply_offered {
335         auction.supply_offered = supply_offered.into();
336     }
337     if let Some(start_timestamp) = start_timestamp {
338         auction.start_timestamp = start_timestamp;
339     }
340     if let Some(end_timestamp) = end_timestamp {
341         auction.end_timestamp = end_timestamp;
342     }
343     if let Some(start_price) = start_price {
344         auction.start_price = start_price.into();
345     }
346     if let Some(end_price) = end_price {
```

```
347         auction.end_price = end_price.into();
348     }
349
350     require!(
351         auction.end_timestamp > auction.start_timestamp,
352         ERR_BAD_START_TIME
353     );
354     require!(auction.end_price < auction.start_price, ERR_BAD_START_PRICE);
355
356     self.auctions.replace(auction_id, &auction);
357
358     auction
359 }
```

**Listing 2.15:** contracts/corn/src/auction.rs

**Suggestion I** Make sure the adjusted `start_timestamp` is larger than the `current_timestamp`.

### 2.3.4 Incompatible Tokens

**Status** Confirmed

**Introduced by** [Version 1](#)

**Description** Elastic supply tokens could dynamically adjust their price, supply, user's balance, etc. For example, inflation tokens, deflation tokens, rebasing tokens, and so forth. In the current implementation of protocol, elastic supply tokens are not supported. If the token is a deflation token, there will be a difference between the recorded amount of transferred tokens to this smart contract (as a parameter of function `ft_on_transfer`) and the actual number of transferred tokens (the token smart contract itself). That's because a small number of tokens will be burned by the token smart contract.

Besides, some of the tokens in [Flux Oracle](#) are not compatible with the protocol as well. For example, for [Linear](#) token, the price provided by [Flux Oracle](#) is not its USD value because the pair is [LiNEAR - NEAR](#). The amount of mintable [Corn](#) tokens will be incorrect if the calculation is based on this pair.

**Suggestion I** Do not use elastic supply tokens, and check whether the pair includes [USD](#) before using its price provided by oracle.

**Feedback from the Project** In the near future we will not use any elastic supply tokens. Note that the project added the patch to ensure that the Flux pair used in contract [corn](#) must contain the USD base token.

### 2.3.5 Timely distribute() upon the Epoch Change

**Status** Confirmed

**Introduced by** [Version 1](#)

**Description** Function `internal_distribute()` is used to distribute the reward of the last [epoch](#) for both the staking and the locking. A few problems could happen if it's not triggered for more than one [epoch](#). For instance, if it's not triggered for one epoch, and a user unstakes to trigger the function at the beginning of a new epoch, the rewards of the last [epoch](#) and the [epoch](#) before the last [epoch](#) would not be distributed to the user.

```
149 pub(crate) fn internal_unstake(&mut self, account_id: &AccountId, num_shares: Shares) {
150     // Ensure enough prepaid gas to transfer $CORN back to account
151     let min_gas =
152         GAS_FOR_FT_TRANSFER + GAS_FOR_UNSTAKE + GAS_FOR_ON_UNSTAKE + GAS_FOR_DISTRIBUTE;
153     require!(
154         env::prepaid_gas() >= min_gas,
155         format!("{}", require_at_least {:?}", ERR_NO_ENOUGH_GAS, min_gas)
156     );
157
158     require!(self.total_staked_corn > 0, ERR_CONTRACT_NO_STAKED_BALANCE);
159     require!(num_shares > 0, ERR_NON_POSITIVE_CALCULATED_UNSTAKING_SHARE);
160     require!(
161         self.internal_ft_balance(account_id) >= num_shares,
162         ERR_NO_ENOUGH_STAKED_BALANCE
163     );
164
165     // Calculating the amount of tokens the account will receive by unstaking the corresponding
166     // number of "stake" shares, rounding down.
167     let receive_amount = self.shares_to_amount(num_shares, false);
168     require!(
169         receive_amount > 0,
170         ERR_NON_POSITIVE_CALCULATED_STAKED_AMOUNT
171     );
172
173     // Distribute rewards if the method is called at the very beginning of the epoch.
174     // If's OK to distribute rewards even if ft_transfer() below fails, so we don't call the
175     // method
176     // in on_unstake()
177     self.internal_distribute();
178
179     // Burn stake shares ($xCORN)
180     self.burn_xcorn(account_id, num_shares, Some("unstake"));
181
182     // Decrease total staked $CORN amount
183     self.total_staked_corn -= receive_amount;
184
185     // Transfer unstaked $CORN to the account
186     // The account data update will be done in the callback
187     ext_fungible_token::ext(self.internal_config().corn_account_id)
188         .with_attached_deposit(1)
189         .with_unused_gas_weight(1)
190         .ft_transfer(
191             account_id.clone(),
192             U128(receive_amount),
193             Some("unstake".to_string()),
194         )
195         .then(
196             Self::ext(env::current_account_id())
197                 .with_unused_gas_weight(2)
198                 .on_unstake(account_id.clone(), U128(receive_amount), U128(num_shares)),
199         );
200 }
```

---

**Listing 2.16:** contracts/xcorn/src/stake.rs

**Suggestion I** It is suggested to invoke the function `distribute()` at the beginning of each `epoch` by team.

**Feedback from the Project** The team will launch a `cron job` that triggers function `distribute()` when every epoch starts. It's also welcome if anyone from community are interested to do this.

### 2.3.6 Potential Centralization Problem

**Status** Confirmed

**Introduced by** [Version 1](#)

**Description** This project has potential centralization problems. The `owner` and the `policy team` have the privilege to configure a number of system parameters. The `owner` even has the ability to upgrade the protocol. Besides, the person who has the private key of allocators could transfer authorized assets from the `treasury` to wherever he/she wants.

**Suggestion I** It is recommended to introduce a decentralization design in the contract, such as a multi-signature or a public DAO.

**Feedback from the Project** Note that the owner and policy team have already been configured as `corndao.sputnikv2.testnet` and `corndao-policy.sputnikv2.testnet`, respectively. When Cornerstone is launched in mainnet, both owner and policy team will be configured as `corndao.sputnik-dao.near` and `corndao-policy.sputnik-dao.near`, which are created by [AstroDAO](#), respectively. The allocator contracts will be locked so there's will be no keys left on the allocator accounts.

### 2.3.7 Missed Sanity Check in `set_stakeholder()`

**Status** Fixed in [Version 2](#)

**Introduced by** [Version 1](#)

**Description** In function `do_epoch_distribute()`, the `stakeholder_total_weight` will be checked in lines 381-384. If it is larger than `FULL_BASIS_POINT/3`, the function will be reverted.

```
328 fn do_epoch_distribute(  
329     &mut self,  
330     staking_mint_amount: Balance,  
331     xcorn_locked: Balance,  
332     xcorn_supply: Balance,  
333     epoch: Epoch,  
334 ) {  
335     let epoch_height = epoch.height;  
336  
337     let bonder_supply_growth = self.accum_bonder_mint_amount;  
338     self.accum_bonder_mint_amount = 0;  
339  
340     let staker_supply_growth = staking_mint_amount;  
341  
342     // ve(3,3) formula: CORNlocker = (1 - xCORNlock/xCORNSupply) * CORNstaker  
343     // Corner cases that CORNlocker should be 0, because nothing is locked:
```

```
344 // 1. If CORNstaker (staked CORN * reward rate) or xCORNSupply is 0, it means no one has
    // staked, so no one can do lock
345 // 2. If xCORNlock is 0 (no xCORN is locked), it means no one are locking
346 let locker_supply_growth =
347     if staker_supply_growth != 0 && xcorn_supply != 0 && xcorn_locked != 0 {
348         staker_supply_growth
349         - (BigDecimal::from(staker_supply_growth) * BigDecimal::from(xcorn_locked)
350           / BigDecimal::from(xcorn_supply))
351         .round_u128()
352     } else {
353         0_u128
354     };
355
356 let community_supply_growth =
357     bonder_supply_growth + staker_supply_growth + locker_supply_growth;
358
359 let mut stakeholders = self.get_stakeholders();
360 let stakeholders_vec = vec![
361     &mut stakeholders.dao,
362     &mut stakeholders.team,
363     &mut stakeholders.seed_round_investors,
364     &mut stakeholders.strategic_round_investors,
365     &mut stakeholders.advisors,
366 ];
367 let stakeholder_names = vec![
368     "dao",
369     "team",
370     "seed round investors",
371     "strategic round investors",
372     "advisors",
373 ];
374
375 let stakeholder_total_weight = stakeholders_vec
376     .iter()
377     .map(|item| item.weight)
378     .reduce(|sum, weight| sum + weight)
379     .unwrap_or(0);
380
381 require!(
382     stakeholder_total_weight <= FULL_BASIS_POINT / 3,
383     ERR_STAKEHOLDER_TOTAL_WEIGHT_HIGH
384 );
385
386 // total supply growth = 100% = community supply growth + stakeholder supply growth
387 let total_supply_growth = community_supply_growth * FULL_BASIS_POINT as u128
388     / (FULL_BASIS_POINT - stakeholder_total_weight) as u128;
```

**Listing 2.17:** contracts/corn/src/treasury.rs

However, there is no such check in the initialization of the stakeholders.

```
73 pub fn set_stakeholder(
74     &mut self,
75     dao: Option<StakeholderParam>,
```

```
76     team: Option<StakeholderParam>,
77     seed_round_investors: Option<StakeholderParam>,
78     strategic_round_investors: Option<StakeholderParam>,
79     advisors: Option<StakeholderParam>,
80 ) {
81     self.assert_owner();
82
83     let params = vec![
84         &dao,
85         &team,
86         &seed_round_investors,
87         &strategic_round_investors,
88         &advisors,
89     ];
90     params.iter().for_each(|o| {
91         if let Some(param) = o {
92             require!(
93                 param.weight <= FULL_BASIS_POINT / 5,
94                 ERR_SINGLE_STAKEHOLDER_WEIGHT
95             );
96         }
97     });
98
99     if let Some(dao_param) = &dao {
100         require!(
101             dao_param.max_distribution.0 == 0,
102             ERR_DAO_SHOULD_HAVE_NO_LIMIT
103         );
104     }
105
106     // the first time this function is called, all params are required
107     if self.stakeholders.is_none() {
108         require!(params.iter().all(|p| p.is_some()), ERR_MISSING_STAKEHOLDERS);
109
110         let stakeholders = Stakeholders {
111             dao: StakeholderInfo::new(dao),
112             team: StakeholderInfo::new(team),
113             seed_round_investors: StakeholderInfo::new(seed_round_investors),
114             strategic_round_investors: StakeholderInfo::new(strategic_round_investors),
115             advisors: StakeholderInfo::new(advisors),
116         };
117
118         self.stakeholders.set(&stakeholders);
119
120         return;
121     }
122
123     let mut stakeholders = self.get_stakeholders();
124
125     if let Some(param) = dao {
126         stakeholders.dao.account_id = param.account_id;
127         stakeholders.dao.weight = param.weight;
128         // dao should have no distribution limitation
```

```
129     }
130     if let Some(param) = team {
131         stakeholders.team.account_id = param.account_id;
132         stakeholders.team.weight = param.weight;
133         stakeholders.team.max_distribution = param.max_distribution.into();
134     }
135     if let Some(param) = seed_round_investors {
136         stakeholders.seed_round_investors.account_id = param.account_id;
137         stakeholders.seed_round_investors.weight = param.weight;
138         stakeholders.seed_round_investors.max_distribution = param.max_distribution.into();
139     }
140     if let Some(param) = strategic_round_investors {
141         stakeholders.strategic_round_investors.account_id = param.account_id;
142         stakeholders.strategic_round_investors.weight = param.weight;
143         stakeholders.strategic_round_investors.max_distribution = param.max_distribution.into();
144     };
145     if let Some(param) = advisors {
146         stakeholders.advisors.account_id = param.account_id;
147         stakeholders.advisors.weight = param.weight;
148         stakeholders.advisors.max_distribution = param.max_distribution.into();
149     }
150
151     self.stakeholders.set(&stakeholders);
152 }
```

**Listing 2.18:** contracts/corn/src/manage.rs

**Suggestion I** Add the sanity check to make sure the sum of weight of all members is smaller or equal to `FULL_BASIS_POINT/3` in function `set_stakeholder()`.

### 2.3.8 Missed `assert_one_yocto()` in System Configuration

**Status** Fixed in [Version 2](#)

**Introduced by** [Version 1](#)

**Description** As shown in the privileged function `set_owner()` below, it requires the `owner` to attach one yocto to invoke the function. However, there is no such requirement for privileged function `set_policy_team()`, function `config()`, and function `set_stakeholder()`, which is not consistent.

```
56 #[payable]
57 pub fn set_owner(&mut self, owner_id: AccountId) {
58     assert_one_yocto();
59     self.assert_owner();
60     self.owner_id = owner_id;
61 }
62
63 pub fn set_policy_team(&mut self, policy_team_id: AccountId) {
64     self.assert_owner();
65     self.policy_team_id = policy_team_id;
66 }
67
68 pub fn config(&mut self, config: Config) {
```

```
69     self.assert_owner();
70     self.config.set(&config);
71 }
72
73 pub fn set_stakeholder(
74     &mut self,
75     dao: Option<StakeholderParam>,
76     team: Option<StakeholderParam>,
77     seed_round_investors: Option<StakeholderParam>,
78     strategic_round_investors: Option<StakeholderParam>,
79     advisors: Option<StakeholderParam>,
80 ) {
81     self.assert_owner();
82
83     let params = vec![
84         &dao,
85         &team,
86         &seed_round_investors,
87         &strategic_round_investors,
88         &advisors,
89     ];
90     params.iter().for_each(|o| {
91         if let Some(param) = o {
92             require!(
93                 param.weight <= FULL_BASIS_POINT / 5,
94                 ERR_SINGLE_STAKEHOLDER_WEIGHT
95             );
96         }
97     });
98
99     if let Some(dao_param) = &dao {
100         require!(
101             dao_param.max_distribution.0 == 0,
102             ERR_DAO_SHOULD_HAVE_NO_LIMIT
103         );
104     }
105
106     // the first time this function is called, all params are required
107     if self.stakeholders.is_none() {
108         require!(params.iter().all(|p| p.is_some()), ERR_MISSING_STAKEHOLDERS);
109
110         let stakeholders = Stakeholders {
111             dao: StakeholderInfo::new(dao),
112             team: StakeholderInfo::new(team),
113             seed_round_investors: StakeholderInfo::new(seed_round_investors),
114             strategic_round_investors: StakeholderInfo::new(strategic_round_investors),
115             advisors: StakeholderInfo::new(advisors),
116         };
117
118         self.stakeholders.set(&stakeholders);
119
120         return;
121     }
```



```
122
123     let mut stakeholders = self.get_stakeholders();
124
125     if let Some(param) = dao {
126         stakeholders.dao.account_id = param.account_id;
127         stakeholders.dao.weight = param.weight;
128         // dao should have no distribution limitation
129     }
130     if let Some(param) = team {
131         stakeholders.team.account_id = param.account_id;
132         stakeholders.team.weight = param.weight;
133         stakeholders.team.max_distribution = param.max_distribution.into();
134     }
135     if let Some(param) = seed_round_investors {
136         stakeholders.seed_round_investors.account_id = param.account_id;
137         stakeholders.seed_round_investors.weight = param.weight;
138         stakeholders.seed_round_investors.max_distribution = param.max_distribution.into();
139     }
140     if let Some(param) = strategic_round_investors {
141         stakeholders.strategic_round_investors.account_id = param.account_id;
142         stakeholders.strategic_round_investors.weight = param.weight;
143         stakeholders.strategic_round_investors.max_distribution = param.max_distribution.into()
144         ;
145     }
146     if let Some(param) = advisors {
147         stakeholders.advisors.account_id = param.account_id;
148         stakeholders.advisors.weight = param.weight;
149         stakeholders.advisors.max_distribution = param.max_distribution.into();
150     }
151     self.stakeholders.set(&stakeholders);
152 }
```

**Listing 2.19:** contracts/corn/src/manage.rs

**Suggestion I** It's suggested to add the function `assert_one_yocto()` in them for consistency and safety since they are all sensitive privileged functions.

## 2.4 Notes

### 2.4.1 Delayed Price from Oracle

**Status** Confirmed

**Introduced by** [version 1](#)

**Description** Given the async nature of NEAR protocol, one transaction on NEAR protocol may be executed in several blocks. Therefore, the price returned back from the oracle might not be the latest. Meanwhile, the max delay time for price is 5 minutes, which should be noted.

**Feedback from the Project** We're aware of this. Flux oracle and LPT oracle created by us will try to keep the prices up-to-date. In practice, prices in Flux oracle and LPT oracle are updated if deviation exceeds

±0.5%, with a minimum update interval set to 1 hour (heartbeat update). So we'll set the max acceptable delay time to 60 minutes to work correctly with Flux.

## 2.4.2 Inconsistency of Valuation between `corn_lp_token` and `general_lp_token`

**Status** Confirmed

**Introduced by** [Version 1](#)

**Description** The calculation of the value between `corn_lp_token` and `general_lp_token` is inconsistent. The value of the `corn_lp_token` is calculated as:

$$\frac{2 * \sqrt{reserve1 * reserve2} * \sqrt{price1 * price2}}{total\_supply}$$

```
25pub(crate) fn valuate_corn_lp_token(
26    lp_token_amount: Balance,
27    lp_token_total_supply: Balance,
28    pool_corn_token_amount: Balance,
29    pool_other_token_amount: Balance,
30    other_token_decimals: u8,
31    other_token_unit_price: Balance,
32    price_decimals: u8,
33) -> Balance {
34    // origin equation is:
35    // BigDecimal::from(other_token_unit_price)
36    //     * BigDecimal::from(pool_corn_token_amount)
37    //     * BigDecimal::from(pool_other_token_amount)
38    //     / decimals(CORN_DECIMALS + other_token_decimals + price_decimals);
39    //
40    // however, since we want to make the final result in CORN decimals, and there is a sqrt later
41    // so we need to multiply the value above with 2*CORN_DECIMALS, which leads to:
42    let pk = BigDecimal::from(other_token_unit_price) * BigDecimal::from(pool_other_token_amount)
43        / decimals(price_decimals + other_token_decimals)
44        * BigDecimal::from(pool_corn_token_amount)
45        * decimals(CORN_DECIMALS);
46
47    let val = BigDecimal::from(2_u64) * pk.sqrt() * BigDecimal::from(lp_token_amount)
48        / BigDecimal::from(lp_token_total_supply);
49
50    val.round_u128()
51}
```

**Listing 2.20:** `contracts/corn/src/bonding/valuation.rs`

The value of the `general_lp_token` is calculated as:

$$\frac{reserve1 * price1 + reserve2 * price2}{total\_supply}$$

It's suggested to calculate the value in the same way to make the valuation consistent and more fair.

```
56 pub(crate) fn valuate_general_lp_token(
57     lp_token_amount: Balance,
58     lp_token_total_supply: Balance,
59     token0_amount: Balance,
```

```
60 token1_amount: Balance,
61 token0_decimals: u8,
62 token1_decimals: u8,
63 token0_unit_price: Balance,
64 token1_unit_price: Balance,
65 price_decimals: u8,
66) -> Balance {
67   let token0_total_value = BigDecimal::from(token0_amount)
68     * BigDecimal::from(token0_unit_price)
69     * decimals(CORN_DECIMALS)
70   / decimals(token0_decimals + price_decimals);
71   let token1_total_value = BigDecimal::from(token1_amount)
72     * BigDecimal::from(token1_unit_price)
73     * decimals(CORN_DECIMALS)
74   / decimals(token1_decimals + price_decimals);
75
76   ((token0_total_value + token1_total_value) * BigDecimal::from(lp_token_amount)
77     / BigDecimal::from(lp_token_total_supply))
78   .round_u128()
79}
```

**Listing 2.21:** contracts/corn/src/bonding/valuation.rs

**Feedback from the Project** This is designed like this. LP tokens that contain CORN have different meaning to the treasury than normal LP tokens.

### 2.4.3 Unrestricted Staking Duration

**Status** Confirmed

**Introduced by** [Version 1](#)

**Description** For staking, rewards are distributed epoch by epoch, and the amount of rewards for each epoch are calculated based on the `total_staked_corn` of the last epoch. However, the minimum staking time is not restricted in staking, which allows users to stake before the end of the last period and to unstake after the start of the new period. In this case, users can earn the same amount of rewards as those who stake for the whole period. It's suggested to set a minimum staking duration.

```
108 pub(crate) fn internal_stake(&mut self, account_id: &AccountId, amount: Balance) {
109   let min_gas = GAS_FOR_STAKE + GAS_FOR_DISTRIBUTE;
110   require!(
111     env::prepaid_gas() >= min_gas,
112     format!("{}", require at least {:?}, ERR_NO_ENOUGH_GAS, min_gas)
113   );
114
115   require!(amount > 0, ERR_NON_POSITIVE_STAKING_AMOUNT);
116
117   // Calculate the number of "stake" shares that the account will receive for staking the
118   // given amount.
119   let num_shares = self.amount_to_shares(amount, false);
120   require!(num_shares > 0, ERR_NON_POSITIVE_CALCULATED_STAKING_SHARE);
121
122   // Distribute rewards if the method is called at the very beginning of the epoch
```

```
123     self.internal_distribute();
124
125     // Mint $xCORN for the account
126     self.mint_xcorn(account_id, num_shares, Some("stake"));
127
128     // Increase total staked $CORN
129     self.total_staked_corn += amount;
130     // Increase total staked amount at current epoch
131     self.staked_corn_tracker
132         .increase_current_epoch_balance(self.internal_distributor().epoch, amount);
133
134     Event::Stake {
135         account_id,
136         staked_amount: &U128(amount),
137         minted_stake_shares: &U128(num_shares),
138         new_stake_shares: &U128(self.internal_ft_balance(account_id)),
139     }
140     .emit();
141     log!(
142         "Contract total staked balance is {}. Total number of shares {}",
143         self.total_staked_corn,
144         self.xcorn_total_supply()
145     );
146 }
```

**Listing 2.22:** contracts/xcorn/src/stake.rs

```
107 pub(crate) fn internal_next_rewards(&self) -> Balance {
108     let total_staked_corn = self.total_staked_amount_at_epoch_start();
109     (U256::from(self.internal_config().reward_rate) * U256::from(total_staked_corn)
110      / U256::from(RATE_DENOMINATOR))
111     .as_u128()
112 }
```

**Listing 2.23:** contracts/xcorn/src/distributor.rs

**Feedback from the Project** This is a known potential arbitrage behavior, but we confirmed that there's no benefit if someone is acting in this way.