ackee blockchain

1inch Farming

9 February 2022

by Ackee Blockchain

Table of Contents

Document Revisions

Revision	Date	Description
1.0	28 Jan 2022	Initial revision
1.1	9 Feb 2022	Developers' feedback

1. Overview

This document presents our findings in reviewed contracts.

1.1 Ackee Blockchain

Ackee Blockchain is an auditing company based in Prague, Czech Republic, specialized in audits and security assessments. Our mission is to build a stronger blockchain community by sharing knowledge – we run a free certification course Summer School of Solidity and teach at the Czech Technical University in Prague. Ackee Blockchain is backed by the largest VC fund focused on blockchain and DeFi in Europe, Rockaway Blockchain Fund.

1.2 Audit Methodology

1. **Technical specification/documentation** - a brief overview of the system is requested from the client and the scope of the audit is defined.
2. **Tool-based analysis** - deep check with automated Solidity analysis tools is performed.
3. **Manual code review** - the code is checked line by line for common vulnerabilities, code duplication, best practices and the code architecture is reviewed.
4. **Local deployment + hacking** - contracts are deployed locally and we try to attack the system and break it.
5. **Unit testing** - run unit tests to ensure that the system works as expected, potentially we write our own unit tests for specific suspicious scenarios.

1.3 Review team

The audit has been performed with a total time donation of 10 engineering days. The work has been divided between two auditors. The whole process was supervised by the Audit Supervisor.

Member's Name	Position
Štěpán Šonský	Lead Auditor
Lukáš Böhm	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

1.4 Disclaimer

We've put our best effort to find all vulnerabilities in the system, however our findings shouldn't be considered as a complete list of all existing issues.

2. Scope

This chapter describes the audit scope, contains provided specification, used documentation and set main objectives for the audit process.

2.1 Coverage

Files being audited:

- accounting
  - FarmAccounting.sol
  - UserAccounting.sol
- BaseFarm.sol
- ERC20Farmable.sol
- Farm.sol
- FarmingPool.sol

Sources revision used during the whole auditing process:

- Repository: **https://github.com/1inch/farming**
- Commit: **d191cc14b526e2eab09c266580b84116b2630da8**

2.2 Supporting Documentation

We've used the official README which contains a brief description of the system.

- **https://github.com/1inch/farming/blob/master/README.md**

"This repository offers 2 ways to have farming (incentives). Highly recommend to use second option for pools/share/utility tokens by deriving them from ERC20Farmable smart contract. If it's too late you should consider first option as well:

1. FarmingPool.sol offers smart contract where you can stake/deposit specific tokens to get continiously distributed rewards.
2. ERC20Farmable.sol allows derived tokens to have farming without necessarity to stake/deposit token into pool. Moreover it allows to have multiple farmings simultaneously and setup new farms permissionlessly."

2.3 Objectives

We've defined following main objectives of the audit:

- Review the code quality, architecture and best practices.
- Check for vulnerabilities if nobody is able to steal funds or damage contracts.
- Validate algorithms and math calculations for misbehaviors.
- Check if the contracts' owner is not overpowered.

3. System Overview

This chapter describes the audited system from our understanding. The whole project is based on the HardHat development framework and uses Solidity 0.8.x which handles integer underflow/overflow. There are no `unchecked` or `assembly` blocks in the project.

As mentioned in the official readme, there are 2 implemented approaches of farming:

- `FarmingPool` - Based on `BaseFarm` and `ERC20` and uses a logic from `FarmAccounting` and `UserAccounting` libraries.
- `ERC20Farmable` - Based on `ERC20` and uses the `UserAccounting` library and also uses `IFarm` interface for communication with farms.

BaseFarm.sol

Following chapter describes our detailed understanding of the `BaseFarm.sol` contract and its parts.

The contract uses OpenZeppelin library and inherits from `Ownable` and uses `SafeERC20` library for `IERC20`.

Functions

```
constructor(IERC20 stakingToken_, IERC20 rewardsToken_)
```

- Inits staking and reward tokens

```
setDistributor(address distributor_) external onlyOwner
```

- Sets the distributor (only the owner can do that)

```
startFarming(uint256 amount, uint256 period) external
```

- Only the `distributor` can start farming
- Transfers `amount` of reward tokens from `msg.sender` to the contract
- Updates farming state using virtual function `_updateFarmingState()`
- Starts farming on the `farmInfo`

```
_updateFarmingState() internal virtual
```

- Empty virtual function

ERC20Farmable.sol

Following chapter describes our detailed understanding of the `ERC20Farmable.sol` contract and its parts.

The contract uses OpenZeppelin library and inherits from `IERC20Farmable` and `ERC20`.

Functions

`farmBalanceOf(address farm_, address account) public view returns (uint256)`

- Returns balance of the farm if exists otherwise 0

`userFarms(address account) external view returns(address[] memory)`

- Returns an address array of user farms

`farmedPerToken(address farm_) public view returns (uint256 fpt)`

- Calls and returns result of `farmedPerToken()` on specified farm

`farmed(address farm_, address account) external view returns (uint256)`

- Calls and returns result of `farmed()` on specified farm

`farm(address farm_) external override`

- Reverts if farm exists
- Updates balances on the farm
- Increments farm's total supply by `msg.sender`'s `balance`

`exit(address farm_) external override`

- Reverts if farm already exited
- Updates balances on the farm
- Decrements farm's total supply by `msg.sender`'s `balance`

`claim(address farm_) external override`

- Calculates the amount from `farmedPerToken()` and `msg.sender`'s balance on the farm
- If `amount > 0` calls `eraseFarmed()` and `claimFor()` functions on the farm

`checkpoint(address farm_) external override`

- Calls `checkpoint()` on the farm

- Tries to call `farmingCheckpoint()` on the farm
- In case of failure emits the `Error` event

`_beforeTokenTransfer(address from, address to, uint256 amount) internal override`

- Calls `super._beforeTokenTransfer()` (OpenZeppelin's `ERC20`)
- If amount is > 0 then does following
- Iterates through user farms `from` and `to`
- If both parties are farming the same token, then updates balances on the farm with `inFrom` and `inTo` params both `true`
- If sender is farming a token and receiver not, then updates balances on the farm with `inFrom = ture` and `inTo = false`
- If sender isn't farming a token and receiver yes, then updates balances on the farm with `inFrom = false` and `inTo = true`

`_getTotalSupply(address farm_) internal view returns(uint256)`

- Returns total supply of the farm

`_getFarmedSinceCheckpointScaled(address farm_, uint256 updated) internal view returns(uint256)`

- Tries to get farmed amount since checkpoint

Farm.sol

Following chapter describes our detailed understanding of the `Farm.sol` contract and its parts.

The contract inherits from `IFarm` and `BaseFarm`.

Functions

`constructor(IERC20 stakingToken_, IERC20 rewardsToken_)`

- Forwards parameters to `BaseFarm` constructor

`farmingCheckpoint() public override`

- Calls `farmInfo.farmingCheckpoint()`

`farmedSinceCheckpointScaled(uint256 updated) public view override returns(uint256 amount)`

- Return the result of `farmInfo.farmedSinceCheckpointScaled()`

`claimFor(address account, uint256 amount) external override`

- Requires `msg.sender` to be the `stakingToken`
- Transfers `amount` of `rewardToken` from the contract to the `account`

`_updateFarmingState() internal override`

- Calls `stakingToken.checkpoint()`

FarmingPool.sol

Following chapter describes our detailed understanding of the `FarmingPool.sol` contract and its parts.

FarmingPool uses `ERC20` and `IERC20Metadata` from OpenZeppelin library and inherits from `IFarmingPool`, `BaseFarm` and `ERC20`.

Functions

`constructor(IERC20Metadata stakingToken_, IERC20 rewardsToken_)`

- Forwards parameters to `BaseFarm` constructor
- Calls ERC20 constructor with encoded `stakingToken` `name` and `symbol`

`decimals() public view override returns (uint8)`

- Returns `stakingToken` decimals

`farmedPerToken() public view override returns (uint256)`

- Returns the result of `userInfo.farmedPerToken()`

`farmed(address account) public view override returns (uint256)`

- Returns the result of `userInfo.farmed()`

`deposit(uint256 amount) external override`

- Mints pool token `amount` to `msg.sender`
- Transfers `amount` of staking token from the `msg.sender` to the pool

`withdraw(uint256 amount) public override`

- Burns pool token `amount` from the `msg.sender`
- Transfers `amount` of staking token from the pool to the `msg.sender`

`claim() public override`

- If user's farmed amount is greater than `0`, then does following
- Calls `userInfo.eraseFarmed()`
- Transfers farmed amount of the reward token from the pool to the `msg.sender`

`exit() public override`

- Withdraws all `msg.sender`'s funds from the pool

`_beforeTokenTransfer(address from, address to, uint256 amount) internal override`

- Calls `super._beforeTokenTransfer()`
- If `amount` is greater than `0`, calls `userInfo.updateBalances()`

`_getTotalSupply(address /* farm */) internal view returns(uint256)`

- Returns total supply

`_getFarmedSinceCheckpointScaled(address /* farm */, uint256 updated) internal view returns(uint256)`

- Calls `farmInfo.farmedSinceCheckpointScaled()`

`_updateFarmingState() internal override`

- Calls `checkpoint()` on `userInfo`
- Calls `farmInfo.farmingCheckpoint()`

FarmAccounting.sol

Following chapter describes our detailed understanding of the `FarmingAccounting.sol` contract and its parts.

FarmAcounting is a Solidity library, which uses OpenZeppelin Math library.

Structures

`Info`

- `uint40 finished`
- `uint40 duration`
- `uint176 reward`

Functions

`farmingCheckpoint(Info storage info) internal`

- Empty function

`farmedSinceCheckpointScaled(Info storage info, uint256 updated) internal view returns(uint256 amount)`

- Calls `_farmedSinceCheckpointScaledMemory()`

`_farmedSinceCheckpointScaledMemory(Info memory info, uint256 updated) private view returns(uint256 amount)`

- If `info.duration` is greater than 0, then does following
- Calculates elapsed time subtracting:
  - `Math.min(block.timestamp, info.finished)` and
  - `Math.max(updated, info.finished - info.duration)`
- Returns farmed reward tokens since checkpoint

`startFarming(Info storage info, uint256 amount, uint256 period) internal returns(uint256)`

- If `block.timestamp` is before `info.finished` then `amount` in incremented by `reward` minus farmed amount since checkpoint
- Checks `period` and `amount` conditions using `require` statement
- Recalculates `info.finished`, `info.duration` and `info.reward`
- Returns amount

UserAccounting.sol

Following chapter describes our detailed understanding of the `UserAccounting.sol` contract and its parts.

`UserAccounting` is a Solidity library, which doesn't have any dependencies.

Structures

`Info`

- `uint40 updateTime`
- `uint216 farmedPerTokenStored`
- `mapping(address => int256) corrections`

Functions

`farmedPerToken(Info storage info, address farm, function(address) internal view returns(uint256) getSupply, function(address, uint256) internal view returns(uint256) getFarmedSinceCheckpointScaled) internal view returns(uint256)`

- If `block.timestamp != info.updateTime` then
- Gets farm's supply using passed `getSupply` function
- If `supply` is greater than `0` then increments `farmedPerTokenStored`

- Returns `farmedPerTokenStored`

`farmed(Info storage info, address account, uint256 balance, uint256 fpt) internal view returns(uint256)`

- Returns farmed amount

`eraseFarmed(Info storage info, address account, uint256 balance, uint256 fpt) internal`

- Sets `info.corrections[account]` to `balance * fpt`

`checkpoint(Info storage info, uint256 fpt) internal`

- If `block.timestamp` not equals to `info.updateTime` or `fpt` not equals to `info.farmedPerTokenStored` then updates values in `info`

`updateBalances(Info storage info, uint256 fpt, address from, address to, uint256 amount, bool inFrom, bool inTo) internal`

- If `amount` is greater than `0` and `inFrom` or `inTo` is `true` then does following
- If `inFrom` or `inTo` is `false` then calls `checkpoint()`
- If inFrom is true then decrements `info.corrections[from]` by farmed amount
- If inTo is true then inccrements `info.corrections[to]` by farmed amount

4.Security Specification

This section specifies single roles and their relationships in terms of security in our understanding of the audited system.

4.1 Actors

This part describes actors of the system, their roles and permissions.

Owner

Owner is the external address which deploys contracts. In the audited system the owner has no special rights except setting a distributor in `BaseFarm` contract.

Distributor

Distributor is the only one who can call the `startFarming()` in the `BaseFarm` contract.

Farm

All the logic and token manipulations are handled by the farming contracts. Only them have these privileges when appropriate users call them.

Users

Users are meant as external addresses which can interact with contracts. They can read information about farmed tokens, deposit, withdraw, claim for reward or exit farming pools.

4.2 Trust model

In the first approach to farming a distributor action is necessary to start farming. Next steps are in the user's hands only and there is no space for malicious manipulation by any single entity.

Trust model of contracts is very well built with no unnecessary privileges given to any specific actor. Users have to put trust just in the contracts' logic.

5. Findings

This chapter shows detailed output of our analysis and testing.

5.1 General Comments

The code quality is excellent. The architecture is well designed and developers are following best practices, using inheritance wisely and the code doesn't contain any duplications. There are used advanced programming techniques like passing function reference as function parameter, which proves developers' seniority.

5.2 Issues

Using our toolset, manual code review and unit testing we've identified the following issues.

Low

Low severity issues are more comments and recommendations rather than security issues. We provide hints on how to improve code readability and follow best practices. Further actions depend on the development team decision.

ID	Description	Contract	Line	Status
L1	Floating pragma	All		Acknowledged
L2	Different compiler versions	FarmingPool.sol	3	Fixed

L1: All contracts are using floating pragma. It's a good practice to use a fixed compiler version.

L1 (rev. 1.1): Developers' acknowledged the issue: "We position this code as a library, so floating pragma is left intentionally."

L2: `FarmingPool.sol` uses `^0.8.9`, other contracts `^0.8.0`. We recommend using the same compiler version across all contracts.

L2 (rev 1.1): Fixed in commit [783fccc647d1a30833d751f6f13100662ce6199f]

Medium

Medium severity issues aren't security vulnerabilities, but should be clearly clarified or fixed.

✓ We haven't found any medium severity issues.

High

High severity issues are potential security vulnerabilities, which require specific steps and conditions to be exploited. Or bugs in the contract logic which doesn't endanger user funds. These issues have to be fixed.

✓ We haven't found any high severity issues.

Critical

Direct critical security threats, which could be instantly misused to attack the system. Or critical bugs in the logic, which leads to the contract misbehavior or users' funds loss. These issues have to be fixed.

✓ We haven't found any critical severity issues.

5.3 Unit testing

Statements test coverage of nearly all contracts is 100%, only `ERC20Farmable.sol` has slightly lower coverage 97,78% which is very good.

`npm run coverage` output:

```
----------------------|----------|----------|----------|----------|----------------|
File                  |  % Stmts | % Branch |  % Funcs |  % Lines |Uncovered Lines |
----------------------|----------|----------|----------|----------|----------------|
 contracts/           |    98.75 |    79.17 |      100 |    98.77 |                |
  BaseFarm.sol        |      100 |      100 |      100 |      100 |                |
  ERC20Farmable.sol   |    97.78 |    81.25 |      100 |    97.83 |             68 |
  Farm.sol            |      100 |      100 |      100 |      100 |                |
  FarmingPool.sol     |      100 |       50 |      100 |      100 |                |
 contracts/accounting/|      100 |    90.91 |      100 |      100 |                |
  FarmAccounting.sol  |      100 |      100 |      100 |      100 |                |
  UserAccounting.sol  |      100 |    85.71 |      100 |      100 |                |
 contracts/interfaces/|      100 |      100 |      100 |      100 |                |
  IERC20Farmable.sol  |      100 |      100 |      100 |      100 |                |
  IFarm.sol           |      100 |      100 |      100 |      100 |                |
  IFarmingPool.sol    |      100 |      100 |      100 |      100 |                |
----------------------|----------|----------|----------|----------|----------------|
All files             |    99.08 |    84.78 |      100 |    99.09 |                |
----------------------|----------|----------|----------|----------|----------------|
```

There are 43 unit tests in the project and all are passing. Tests are covering quite a lot of different scenarios and interactions with the farming platform.

`npx hardhat test` output:

```
Contract: ERC20Farmable
  startFarming
    ✓ should thrown with rewards distribution access denied
  farm
    ✓ should update totalSupply
    ✓ should make totalSupply to decrease with balance
    ✓ should make totalSupply to increase with balance
    ✓ should make totalSupply ignore internal transfers
    ✓ should be thrown
  claimFor
    ✓ should thrown with access denied
  claim
    ✓ should claim tokens
    ✓ should claim tokens for non-user farms wallet
  userFarms
    ✓ should return user farms
  exit
    ✓ should be burn
    ✓ should be thrown
  deposit
    ✓ Two stakers with the same stakes wait 1 w
    ✓ Two stakers with the different (1:3) stakes wait 1 w
    ✓ Two stakers with the different (1:3) stakes wait 2 weeks
    ✓ One staker on 1st and 3rd weeks farming with gap
    ✓ One staker on 1st and 3rd weeks farming with gap + claim in the middle
    ✓ One staker on 1st and 3rd weeks farming with gap + exit/farm in the middle
    ✓ One staker on 1st and 3rd weeks farming with gap + exit/claim in the middle
    ✓ Three stakers with the different (1:3:5) stakes wait 3 weeks
    ✓ One staker on 2 durations with gap
    ✓ Notify Reward Amount from mocked distribution to 10,000
    ✓ Thrown with Period too large
    ✓ Thrown with Amount too large
    ✓ Notify Reward Amount before prev farming finished

Contract: FarmingPool
  startFarming
    ✓ should thrown with rewards distribution access denied
  name
    ✓ should be return name
  symbol
    ✓ should be return symbol
  decimals
    ✓ should be return decimals
  mint
    ✓ should be mint
  burn
    ✓ should be burn
    ✓ should be thrown
  deposit
    ✓ Two stakers with the same stakes wait 1 w
    ✓ Two stakers with the different (1:3) stakes wait 1 w
    ✓ Two stakers with the different (1:3) stakes wait 2 weeks
    ✓ One staker on 1st and 3rd weeks farming with gap
    ✓ One staker on 1st and 3rd weeks farming with gap + claim in the middle
    ✓ Three stakers with the different (1:3:5) stakes wait 3 weeks
    ✓ One staker on 2 durations with gap
    ✓ Notify Reward Amount from mocked distribution to 10,000
    ✓ Thrown with Period too large
    ✓ Thrown with Amount too large
    ✓ Notify Reward Amount before prev farming finished
```

6. Conclusion

We've started with a basic project understanding. Lead auditor defined the audit methodology and objectives. Then we performed a static analysis which didn't identify any serious issues. We have found only minor issues during our intensive manual code review. These issues were related to compiler versions which we categorized as low severity. Since these issues do not pose a security threat we let the developers decide whether to fix these issues or acknowledge them. We didn't identify any attack vectors which could endanger users' funds or the farming platform itself.

Rev. 1.1: We've updated the audit document to revision 1.1 after receiving developers' feedback. L1 issue was acknowledged as intentional and L2 issue was fixed.

ackee blockchain

Thank You

Ackee Blockchain a.s.


Prague, Czech Republic


hello@ackeeblockchain.com


https://discord.gg/z4KDUbuPxq