



Università  
di Genova

DIBRIS DIPARTIMENTO  
DI INFORMATICA, BIOINGEGNERIA,  
ROBOTICA E INGEGNERIA DEI SISTEMI

## Report - Crowdfunding SC 🐔

# **Cock Fight Sim**

---

Decentralized Systems

Master course in Computer Science

Authors:

Cattaneo Kevin - 4944382

Foschi Lorenzo - 4989646

# Index

<b>Index.....</b>	<b>1</b>
<b>I. Crowdfunding Assignment.....</b>	<b>3</b>
<b>II. Implemented Functionalities.....</b>	<b>3</b>
Extra features.....	3
<b>III. Why Cocks?.....</b>	<b>4</b>
<b>IV. The Smart Contract.....</b>	<b>4</b>
The First Smart Contract.....	4
Extra features.....	7
<b>V. Frontend Page.....</b>	<b>10</b>
<b>VI. Proxy or not Proxy, that's the dilemma!.....</b>	<b>10</b>
First version.....	11
Second version.....	12
Third version.....	13
<b>VII. Cocks are secure!.....</b>	<b>13</b>
Remix analysers.....	13
SolidityAnalyzer (BETA) top 5:.....	14
Mythril.....	14
<b>VIII. Appendix.....</b>	<b>15</b>
Setup to run the code.....	15
npm install.....	15
npm start.....	15
npm run build.....	15
Contract addresses.....	15



# I. Crowdfunding Assignment

Design and implement one or more smart contracts and a web interface that mimics an Initial Coin Offering (ICO), where users can receive project Tokens paid in Sepolia ETH.

1. The users of your dApp can pay Sepolia ETH buy your Token and fund your idea.
2. Deploy an ERC-20 contract to create your Token.
3. Implement a soft cap, e.g., the minimum amount of funds that your project aims to raise during the token sale to proceed with its development and goals.
4. Present your idea on your website where users can buy your Tokens by interacting through MetaMask.

## II. Implemented Functionalities

- **Buy tokens:** let users fund our idea by spending Sepolia ETH in exchange for our (CFT) ERC-20 tokens.
- **Finalize** the crowdfunding: let the owner finalize the funding after the deadline set, letting the owner also withdraw the funding if that has reached the soft cap.
- **Claim refund:** if funding has not reached the soft cap and has been finalized, users can be refunded of their money spent

Of course the code of the contract has been verified through Etherscan to allow users to interact with it.

### Extra features

- **Refund the extra:** since we have a token price, if the user spend more than the price times the amount of tokens
- **Pause:** we allow the pause of the contract by the owner if emergency.
- **Early cockers:** we mark a starter period to let users have more tokens with less contribution (we thank our earlier contributors).
- **Leaderboard:** we trace a leaderboard of the best 5 contributors.
- **Proxy** (tentative): built a *read-only* proxy to allow the pointing to a new, better, contract.


### III. Why Cocks?

The idea of the project starts from a real game idea, implemented by us during summer vacations: “Cock Fighting Simulator” 🐔

Despite its courageous naming, the game is on steam ([CFS](#)) and we implemented a real Kickstarter Crowdfunding campaign.

Therefore, when we were prompted with this Smart Contract idea we couldn't help ourselves from choosing that path!

We also produced some slides and a trailer, you can check them at this link:

 Assignment 1 & 3.pdf & [Trailer!](#)

### IV. The Smart Contract

#### The First Smart Contract

For the first step we implemented a basic version of the Smart Contract, containing all the mandatory features, as already reported in Section 2.

```
/*
Constructor sets token details and initial conditions, with following params:
_tokenPrice The price of a single token in wei
_softCap The minimum amount of ETH required for the cock to be successful.
_duration The (seconds) duration for which the cock ICO will remain active.
_earlyBirdBonus The bonus percentage for early birds.
_earlyBirdDuration The duration of the early bird period in seconds.
*/
constructor(uint256 _tokenPrice, uint256 _softCap, uint256 _duration, uint256 _earlyBirdBonus, uint256 _earlyBirdDuration, uint256 _minContribution)
ERC20("CockFundingToken", "CFT")
Ownable(msg.sender)
{
    require(_tokenPrice > 0, "Cock price deserves to be more than zero ETH!");
    require(_softCap > 0, "The Cock project needs more than zero ETH to reach its success :(");
    require(_duration > 0, "The Cock project needs more time than zero seconds to... grow");

    tokenPrice = _tokenPrice;
    softCap = _softCap;
    deadline = block.timestamp + _duration; // start from... now!

    require(_earlyBirdDuration < _duration, "Early bird duration must be shorter than ICO duration");
    earlyBirdBonus = _earlyBirdBonus;
    earlyBirdDuration = _earlyBirdDuration;

    require(_minContribution > 0, "You can't send 0 tokens!");
    minContribution = _minContribution;

    // Mint tokens to the contract for sale (for simplicity, a fixed supply)
    _mint(address(this), 1_069_069 * 10 ** decimals());
}
```

This above is the basic **constructor**: useful to understand which data we handle.

```

// Allows users to buy tokens at the set price until the deadline or finalization.
function buyTokens() public payable whenNotPaused {
    require(!isFinalized, "ICO already finalized!");
    require(block.timestamp <= deadline, "ICO duration is over");
    // No need to check if > 0, because minContribution is already > 0!
    require(msg.value >= minContribution, "Contribution is too small");

    uint256 ethAmount = msg.value;
    uint256 tokenAmount = calculateTokensWithBonus(ethAmount);
    uint256 requiredEth = tokenAmount * tokenPrice; // The Ether equivalent for the tokens they should get

    // Then we have the token logic
    require(tokenAmount > 0, "No Cock for you ^_^");
    require(balanceOf(address(this)) >= tokenAmount, "Holy, u poor");

    // If the user sent more than required, refund the extra Ether
    uint256 refundAmount = ethAmount - requiredEth;

    totalRaised += msg.value;
    contributions[msg.sender] += msg.value;

    updateLeaderboard(msg.sender, contributions[msg.sender]);

    _transfer(address(this), msg.sender, tokenAmount);

    emit TokensPurchased(msg.sender, msg.value, tokenAmount);

    // At the end to avoid reentrancy :D
    if (refundAmount > 0) {
        payable(msg.sender).transfer(refundAmount); // Refund the extra Ether to the sender
    }
}

```

Here above the **buyTokens** is shown:

First, it verifies that the ICO has **not** been **finalized** and that the current time is within the allowed sale period. It also ensures that the contribution meets or exceeds the minimum required amount, thereby preventing purchases that fall below a defined **threshold**.

Once these checks are passed, the function calculates the number of tokens the contributor is eligible to receive based on the ETH they sent, taking into account any early-bird bonuses that might apply (*see part 2 of this section for further details for extra features*).

The function also validates that the contract holds **enough tokens** to fulfill the purchase request. If everything is ok, it updates the total amount of ETH raised and records the contributor's **contribution** to maintain transparency and accountability. The purchased tokens are then transferred to the contributor, and an **event** is emitted to log the transaction.

Finally, if a **refund** is due from any overpayment (we're honest 🍷!), the function ensures the excess ETH is securely returned to the contributor, safeguarding their funds.

```

Allows the owner to finalize the ICO after the deadline.
    If the soft cap is reached or exceeded, ETH is transferred to the owner:
        the Cock project has.. ehm.. is finished!
    If not, contributors can claim refunds:
        the Cock project didn't last for so long...
*/
function finalize() public onlyOwner {
    require(!isFinalized, "Already finalized");
    require(block.timestamp >= deadline, "C'mon we can't finish prematurely!");

    isFinalized = true;
    if (totalRaised >= softCap) {
        // SUCCESS: transfer all raised funds to the owner
        isSuccessful = true;
        uint256 amount = address(this).balance;

        // We pass an empty bytes array to save gas (so we ignore the second output)
        (bool success, ) = owner().call{value: amount}("");
        require(success, "Oh no! Transfer to owner failed");

        emit Withdrawal(owner(), amount);
    } else {
        // FAIL: Contributors will be able to claim refunds :(
        isSuccessful = false;
    }

    emit Finalized(isSuccessful);
}

```

The **finalize** function concludes the ICO after its deadline, ensuring it's **only** callable by the **owner**. It checks that the ICO is not already finalized and that the deadline has passed. If the raised funds meet or exceed the softCap, the ICO is successful, and all ETH collected is transferred to the owner. Otherwise, the ICO fails, and contributors can claim refunds (hopefully this won't happen!). The function sets the **isFinalized** flag, marks success or failure, and emits **events** to log the outcome, ensuring a secure and transparent conclusion.



```

function claimRefund() public {
    require(isFinalized, "Hold up! Not so fast :'(");
    require(!isSuccessful, "ICO was successful, go play with Cocks!");

    uint256 contribution = contributions[msg.sender];
    require(contribution > 0, "No contributions to refund, u don't fool us!");

    // As seen in lesson, let's put the value at zero BEFORE the claim!
    contributions[msg.sender] = 0;

    // For a genoese person this is the worst part
    (bool success, ) = msg.sender.call{value: contribution}("");
    require(success, "Refund transfer failed");

    emit Refunded(msg.sender, contribution);
}

```

Although we would like to run away with all the money, this **claimRefund** allows contributors to claim a refund if the ICO fails. It can be only called **after finalization** and only if softCap wasn't met. Here, to **avoid reentrancy attacks**, it's important to zero the contribution value BEFORE the actual claim... we don't want to be The DAO 2!

```

function withdrawRemainingTokens() external onlyOwner {
    require(isFinalized, "ICO not finalized");
    uint256 remaining = balanceOf(address(this));
    require(remaining > 0, "No remaining tokens");
    _transfer(address(this), owner(), remaining);
}

```

This **withdrawal** function enables the owner to withdraw any leftover tokens after the ICO has ended. It prevents wastage of unused cock tokens, allowing the owner to repurpose them for other initiatives or future sales.

## Extra features

We could stop here, but the Cock project deserves love and attention, and that's why we delved into the extra features already presented in Section 2.1.

We leveraged modifiers to make the contract pausable, implemented an "Early Bids" prize mechanism and a LeaderBoard system!



```

/*
    EMERGENCY! The ICO can be paused.
    While paused, no one can buy tokens.
*/
function pause() external onlyOwner {
    _pause();
}

/*
    Allows the owner to resume the ICO once the emergency is resolved :)
*/
function unpause() external onlyOwner {
    _unpause();
}

/*
    Fallback function to allow direct ETH payment to buy tokens.
    aka when someone sends ETH to this contract without specifying any
    specific function to execute, it will automatically run the receive function
*/
receive() external payable whenNotPaused {
    buyTokens();
}

```

### Pause and Unpause Functions:

- pause: allows the contract owner to pause the ICO during emergencies, preventing any further token purchases.
- unpause: allows the owner to resume the ICO once the issue is resolved.

**Fallback Function (receive):** Automatically triggers the buyTokens function when someone sends ETH directly to the contract without specifying a function call.

```

function calculateTokensWithBonus(uint256 ethAmount) internal view returns (uint256) {
    uint256 tokens = ethAmount / tokenPrice;
    if (block.timestamp <= deadline - earlyBirdDuration) {
        // Apply bonus for the early cockers!
        tokens = (ethAmount * (100 + earlyBirdBonus)) / 100 / tokenPrice;
    }
    return tokens;
}

```

This above function calculates the number of tokens to be awarded based on the provided amount of Ether (the early cockers 🐓!). It takes into account a potential early-bird bonus for contributors who buy tokens before a specified earlyBirdDuration expires.

- **Base Token Calculation:** The number of tokens is determined by dividing the contributed Ether by the token price.
- **Early Bird Bonus:** If the current time (block.timestamp) is within the early-cockers period, an additional percentage bonus is applied. This bonus incentivizes early contributions during the ICO (Initial Coin Offering).

```
// We need to store the top cocks!
function updateLeaderboard(address contributor, uint256 contribution) internal {
    // You! Yes, You! Let's see if you were already here and you added other contributions!
    for (uint256 i = 0; i < leaderboardSize; i++) {
        if (leaderboardAddresses[i] == contributor) {
            leaderboardContributions[i] = contribution;
            sortLeaderboard(); // Sort only if the value has been updated
            emit LeaderboardUpdated(contributor, contribution);
            return;
        }
    }

    // Ok you weren't already here, but let's see if you will!
    uint256 smallestContribution = leaderboardContributions[leaderboardSize - 1];
    if (contribution > smallestContribution) {
        leaderboardAddresses[leaderboardSize - 1] = contributor;
        leaderboardContributions[leaderboardSize - 1] = contribution;
        sortLeaderboard(); // OPT: Sort only if a new contributor enters
        emit LeaderboardUpdated(contributor, contribution);
    }
}

// Naive bubble sort because at the end we only have 5 elements ^^
function sortLeaderboard() internal {
    for (uint256 i = 0; i < leaderboardSize - 1; i++) {
        for (uint256 j = i + 1; j < leaderboardSize; j++) {
            if (leaderboardContributions[i] < leaderboardContributions[j]) {
                (leaderboardAddresses[i], leaderboardAddresses[j]) = (leaderboardAddresses[j], leaderboardAddresses[i]);
                (leaderboardContributions[i], leaderboardContributions[j]) = (leaderboardContributions[j], leaderboardContributions[i]);
            }
        }
    }
}
```

The functions above are meant to build and update the leaderboard of the **top 5 contributors**.

They're meant to be simple, considering the cap at five contributors.



## V. Frontend Page

For the frontend page we wanted to provide a landing page that allowed us to be clear in our objective: **funding the videogame**. So we developed it in React JS in a few paragraphs and some sections that the user can explore to delve into details and interact with the smart contract using ether.js library. In particular:

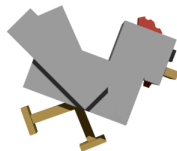
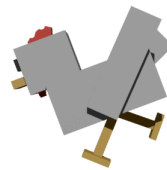
- **Homepage:** represent the first view the user has when navigating to our website, with a few paragraphs that tell more about the project (and the assignment purpose). The style chosen is a simple and direct one, with the modern standard of header and footer, also **responsive** for mobile devices.
- **Funding page:** the core of the interaction with our smart contract, this page trigger the Metamask provider to allow interactions; it is divided in two parts:
  - upper: the contract details, like the funds collected and the token price
  - bottom: the cards that allow you to input values to fund the idea or to retrieve tokens if the project fails after deadline and softCap is not reached
- **Heroes page (Leaderboard):** in this page we put in heading our top 5 contributors
- **Hidden Easter egg page:** we allow users to collect fresh ETH if they are clever enough... maybe.

### Hit that cock!

We are a group of students behind the **Cock Fight Simulator game**. We want to fund our videogame idea to improve the quality!

Buy our tokens to help us!

FUND NOW



### Assignment

This website has been developed as an assignment of the **Decentralized System** master course.

a.y. 2024/25

## VI. Proxy or not Proxy, that's the dilemma!

We also spent A LOT of time trying to develop a proxy that allowed us to point to a new version of our smart contract as definition says. We had a lot of problems with making the delegateCall function work as we wished and in the end we managed to have a fully functional *read-only* proxy, that is a contract that allows us just to read the new information of the contract but not to write in its memory. For each version we reported the screenshot of just the heading part.

## First version

The first version tried to resemble the original contract, so we find all the fields of the original one plus the upgrade function.<sup>4</sup>

```
contract CockfundingProxy is ERC20, Ownable, Pausable {
    address payable public targetContract;

    uint256 public immutable tokenPrice;           // Price of one token in wei (ETH)
    uint256 public totalRaised;                     // Total amount of ETH raised for the cock project
    uint256 public immutable softCap;               // Minimum amount of ETH required for cock success
    uint256 public immutable deadline;              // Timestamp after which cock ICO can be finalized
    uint256 public immutable minContribution;       // To avoid cluttering!

    bool public isFinalized;                        // Has the cock ICO been finalized?
    bool public isSuccessful;                       // Hopefully Cock Fight Sim will meet the soft cap!

    uint256 public immutable earlyBirdBonus;        // Bonus percentage for early birds
    uint256 public immutable earlyBirdDuration;     // Early bird period

    // Tracks ETH contributions for EACH address
    mapping(address => uint256) public contributions;

    // Leaderboard variables (Fixed top 5 contributors!)
    uint256 public constant leaderboardSize = 5;
    address[leaderboardSize] public leaderboardAddresses;
    uint256[leaderboardSize] public leaderboardContributions;

    constructor(address payable _targetContract, uint256 _tokenPrice, uint256 _softCap, uint256 _duration)
        ERC20("CockFundingToken", "CFT")
        Ownable(msg.sender)
    {
        targetContract = _targetContract;

        require(_tokenPrice > 0, "Cock price deserves to be more than zero ETH!");
        require(_softCap > 0, "The Cock project needs more than zero ETH to reach its success :(");
        require(_duration > 0, "The Cock project needs more time than zero seconds to... grow");

        tokenPrice = _tokenPrice;
    }
}
```

and each function performs a delegateCall to the targetContract address:

```
function pause() external onlyOwner {
    targetContract.delegatecall(abi.encodeWithSignature("pause()"));
}

function unpause() external onlyOwner {
    targetContract.delegatecall(abi.encodeWithSignature("unpause()"));
}

function buyTokens() public payable whenNotPaused {
    targetContract.delegatecall(abi.encodeWithSignature("buyTokens()"));
}

function finalize() external onlyOwner {
    targetContract.delegatecall(abi.encodeWithSignature("finalize()"));
}

function claimRefund() external {
    targetContract.delegatecall(abi.encodeWithSignature("claimRefund()"));
}
```

## Second version

This one is the minimalist version that relies only on a delegate function with an assembly fragment to allow data passing, even this version doesn't work well.

```
contract CockfundingProxy is CockfundingTokenStorage {
    address public implementation;

    /**
     * @dev The constructor sets the logic contract and immediately initializes
     * the storage by delegating the call to 'initialize(...)'.
     */
    constructor(
        address _logic,
        uint256 _tokenPrice,
        uint256 _softCap,
        uint256 _duration,
        uint256 _earlyBirdBonus,
        uint256 _earlyBirdDuration,
        uint256 _minContribution,
        address _owner
    ) {
        implementation = _logic;

        // Now call initialize on the logic contract so that our storage is set up
        (bool success, ) = _logic.delegatecall(
            abi.encodeWithSignature(
                "initialize(uint256,uint256,uint256,uint256,uint256,uint256,address)",
                _tokenPrice,
                _softCap,
                _duration,
                _earlyBirdBonus,
                _earlyBirdDuration,
                _minContribution,
                _owner
            )
        );
        require(success, "Initialize failed");
    }
}
```

```
/**
 * @dev The fallback that forwards all calls to the current implementation
 * using DELEGATECALL. The implementation's code executes *using this contract's storage*.
 */
fallback() external payable {
    _delegate();
}

receive() external payable {
    _delegate();
}

function _delegate() internal {
    address impl = implementation;
    assembly {
        // Copy msg.data. We use calldatasize to ensure we get all data.
        calldatacopy(0, 0, calldatasize())

        // Call the implementation.
        // out and outsize are 0 because we don't know the size yet.
        let result := delegatecall(gas(), impl, 0, calldatasize(), 0, 0)

        // Copy the returned data.
        returndatacopy(0, 0, returndatasize())

        switch result
        // delegatecall returns 0 on error.
        case 0 {
            revert(0, returndatasize())
        }
        default {
            return(0, returndatasize())
        }
    }
}
```

## Third version

With this last version we opted for a mix that includes the fallback of the second version and implements the functions by calling them directly on an object of type contract imported from the token implementation, so like calling the functions defined in an external interface.

```
contract CockfundingProxy is Ownable, Pausable {
    address payable public targetContract;

    constructor(address payable _targetContract) Ownable(msg.sender) {
        targetContract = _targetContract;
    }

    function upgrade(address payable _newTargetContract) external onlyOwner {
        targetContract = _newTargetContract;
    }

    fallback() external payable {
        // Forward all calls to the target contract
        address _impl = targetContract;
        assembly {
            calldatacopy(0, 0, calldatasize())
            let result := delegatecall(gas(), _impl, 0, calldatasize(), 0, 0)
            returndatacopy(0, 0, returndatasize())
            switch result
            case 0 { revert(0, returndatasize()) }
            default { return(0, returndatasize()) }
        }
    }

    receive() external payable whenNotPaused {
        return CockfundingToken(targetContract).buyTokens{value: msg.value}();
    }
}
```

We decided to use this last version of proxy, that allowed us to obtain at least a read-only behavior on the new contract's data.

## VII. Cocks are secure!

We also ran some security scans on the token contract in order to address any security issue the scanner would have found. In particular we used both the ones of Remix (Remix proper scanner and Solhint), SolidityScanner (BETA) and Mythril.

Our scanners did not find any critical issue with our code 😊

### Remix analysers

- **Remix:** warnings on block timestamps → used just for set deadline
- **Solhint:** just formatting suggestions of our error messages, no issue signaled

### SolidityAnalyzer (BETA) top 5:

- **INCORRECT ACCESS CONTROL** → false positive, we used Ownable.sol

- **PRECISION LOSS DURING DIVISION BY LARGE NUMBERS** → probability regarding the token computation, but it is not a critical problem if some precision is lost during the transaction  $\text{ETH} \leftrightarrow \text{CFS tokens}$ .
- **USE OWNABLE2STEP** → to avoid owner errors when transferring ownership, but it is not of our interest
- **MISSING EVENTS** → false positive, we put the right and interesting ones
- **OUTDATED COMPILER VERSION** → false positive

## Mythril

Then we put the result of the scanning with Mythril scanner that was 1h long!

```
~/Desktop/ETHgameToken/contracts (main) » time myth a CockfundingToken_flattened.sol  
The analysis was completed successfully. No issues were detected.  
  
myth a CockfundingToken_flattened.sol 3608,88s user 6,71s system 98% cpu 1:01:06,34 total
```

## VIII. Appendix

### Setup to run the code

The website is located in the *crowdfunding* folder, so please position yourself there to run the following.

**npm install**

Runs the installation of dependencies.

**npm start**

Runs the app in the development mode.

**npm run build**

Builds the app for production to the **build** folder.

It correctly bundles React in production mode and optimizes the build for the best performance.

### Contract addresses

- Original contract: 0x538a4aE62e41EfFa33E45FF0e2baDcc4bEa3479A
- Proxy contract: 0x6228A33d9bCf3114E7A9c9333CB5EC2aEE919013
- Upgraded contract: 0xBdc6C79d33FB110E5539E12c37E3A64eFAB12219