DC Report

We decided to refactor the already existing dApp available at AliHaider-codes/Blockchain. The goal was to streamline the architecture, improve the contract, reduce dependencies, and incorporate a more web3-focused approach for storing and retrieving data.

Overview

The platform functions similarly to a Capture The Flag (CTF) competition, where:

- Each challenge provides a description with external resources the user must interact with to find the solution (“flag”).
- Once the user obtains the correct flag signature, they submit it to the **SmartChallenge** contract.
- Verification happens on-chain through ECDSA, and the first user submitting a valid signature receives the reward.
- Since challenge data like its score and the users who have completed it are stored on the blockchain, it is possible to better keep track of player performances and allow them to complete a challenge even if the original ctf website is taken down.

Contract

Contract deployed and verified at: 0×794ECccCD0b6E0eE8526d425f38b36C5EE916744.

Proxy contract deployed and verified at: 0×EC1B8e3c1a34E16765da0D7F1A8676E45fD584b8.

Overview

The **SmartChallenge** contract manages a system of CTF-like challenges, rewards, and players’ scores. Each challenge has an associated **publicFlag** (stored as an address), **reward**, **score**, and a **ipfscid** that points to metadata on IPFS (e.g., challenge name, category, and description). Players can submit a solution (referred to as a “flag”) to earn rewards, which are paid directly from the contract’s balance.

Problems of the previous version

1. **Flags stored on the blockchain**: The flags were stored in the contract, by reading the corresponding **addChallenge** transaction of a challenge you could find its flag.
2. **Frontrunning attack**: It is possible to steal the flag from another player by performing a frontrunning attack.
3. **Sybil attack**: By creating multiple accounts it was possible to eat all the contract credit.
4. **Wasted storage**: Some variables were unused and made the contract price increase (e.g. storing the length of the array in a variable).

Changes from the previous version

- Fix 1. and 2. by using public-key cryptography, the flag to find is a private key which is used to sign a message which is checked by the contract using a public key.
- Fix 3. by giving the reward only to the first solver.
- Fix 4. by reducing the number of variables to the absolute minimum.

Key Components

1. **Challenge Struct**
   - Contains **publicFlag**, **reward**, **score**, and **ipfscid**.
   - **publicFlag** is crucial for verifying the correctness of a submitted flag using signature verification.
2. **Players and Challenges Management**
   - The contract stores which challenges each player has solved.
   - Once a challenge is solved, the contract marks it in an internal mapping to prevent repeated rewards.
3. **Flag Verification**
   - Uses **ECDSA** (**ecrecover**) to verify that the provided signature corresponds to the **publicFlag** address stored in each challenge.
   - A correct submission triggers a reward transfer (**payUser**) and updates the player's solved-challenge records.
   - The reward transaction fee is paid by the user.
4. **Adding New Challenges**
   - Only the owner can add new challenges (with **onlyOwner** modifier).
   - Owner must send an amount of Ether equal to the reward to fund that challenge in the contract.

    - The function to withdraw and deposit money becomes useless since the contract has always the right amount of money to pay for each solved challenge.
5. **Scoring**
    - The contract keeps track of each player's total score by summing the scores of all solved challenges.
    - A simple loop through a player's solved challenge indices is used to compute the total.
    - Since reads from the contracts are free, this is better than the previous version which stored the player's score in a separated variable which was updated every time.

Proxy

Overview

The **SmartChallengeProxy** uses the classic Ethereum Delegate Proxy pattern (eip-897) to separate the contract's storage and logic. This design allows for upgrading the contract implementation without changing the contract's address, preserving state and balances for users.

Key Components

1. **Implementation Address**
    - The proxy stores a **implementation** address, which points to the logic contract (**SmartChallenge**) that handles function calls.
2. **Upgradeability**
    - The **upgradeTo(address _implementation)** function is restricted by **onlyOwner**.
    - This ensures only the contract owner can change the underlying logic contract.
3. **Delegatecall Handling**
    - The **fallback()** and **receive()** functions use **delegatecall** to forward all calls and data to the current **implementation**.
    - The storage layout in the proxy must match that of the logic contract to ensure no data corruption.

Benefits

- **Upgradability**
  If you need to fix bugs or add features to **SmartChallenge**, you can deploy a new version of the logic contract and simply point the proxy to that new implementation.

- **State Preservation**
  Because the proxy never changes its address, you don't have to migrate user balances or challenge data to a new contract address every time you deploy an upgrade.

Frontend

Since all challenge details and flags are stored on the contract (with challenge metadata on IPFS), we no longer need a Next.js server-side layer to interact with them. Instead, we:

- Rewrote the entire frontend using **React** + **TypeScript** + **Radix-UI** components + **Tailwind CSS**.
- Kept the UI look and feel similar to the original dApp but made it more lightweight.
- Retrieve **publicFlag**, **reward**, **score**, and **ipfsHash** from the contract directly.
- Use the **ipfsHash** to fetch additional challenge details (name, description, category) from IPFS.

Since the frontend only accesses data publicly available on the blockchain and on IPFS, the entire website can be statically built and hosted on IPFS making it a complete web3 application.

External databases and backend are needed only for challenges which use the web2 infrastructure.

Infrastructure

All deployment is automatized using docker-compose with the support of a creation of a development environment to avoid testing the contract on the testnet to save Sepolia.

Development environment:

- The Ethereum development network using Hardhat.
- A local IPFS node using kubo.
- Deployment of the smart contract using ape.
- Frontend.

Production environment:

- Sepolia network for smart contract deployment.
- IPFS hosting on Pinata.
- GitHub pages to host the statically built frontend (since Pinata doesn't allow html files).
- fly.io to host the challenges.