DESC 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.

Contract

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 an `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.

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.
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.
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.

Security Considerations

- **Ownership**
  Only the contract owner can add new challenges or perform critical updates.

- **Rewards**
  The contract checks its own balance before paying out rewards to ensure sufficient funds.

- **Signature Verification**
  The contract carefully splits and recovers signatures to ensure correctness of `publicFlag` checks.

Proxy

Overview

The **SmartChallengeProxy** uses the classic Ethereum Proxy pattern (commonly known as the "delegatecall proxy") 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 an `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.

Infra

We realized an automated deployment system using Docker Compose, which launches:

- The Ethereum development network (via **Hardhat**).
- A local IPFS node (instead of relying on third-party pinning services).
- Automated deployment of the SmartChallenge contract onto the development network.
- The frontend.
- Services required for external challenges (such as MySQL).

This setup ensures a fully reproducible development environment where all components are self-contained.

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.

With this approach, the flow is purely web3-based, relying on smart contracts and IPFS alone—no external database or backend needed.

Additionally, we implemented a listener to detect when the user changes their connected MetaMask account. This automatically logs the user out from the previous session for security and clarity.

Challenges

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 users receive rewards if they submit a valid signature.

We also added a SQL injection challenge that must be performed on an external resource. Once a user obtains the correct flag, they submit it via the dApp.

Conclusion

By refactoring the existing dApp and introducing the **SmartChallenge** contract with a dedicated **SmartChallengeProxy**, we have created a more robust, upgradeable, and decentralized application. The new frontend architecture removes the need for a separate backend while preserving a familiar, user-friendly interface. All critical data is now stored or referenced on-chain and via IPFS, ensuring a secure and streamlined CTF-like experience for end users.