Anonymous KYC

About 100 validators are selected through conviction voting to perform KYC. For each user, 5 validators are randomly chosen as jurors. KYC is conducted via end-to-end encrypted P2P chat and video conferencing by selected jurors.

ZKP with validators

Jurors are provided with the user's name, address, and a hash. The user must prove that they know the secret corresponding to the hash.

```rust
use risc0_zkvm::{default_prover, ExecutorEnv};
use sha2::{Digest, Sha256};

let name = "Alice".to_string();
let address = "123 Main St".to_string();
let expiry = 1690000000u64; // Timestamp
let secret = "secret-password"

let env = ExecutorEnv::builder()
    .write(&name)
    .write(&address)
    .write(&secret)
    .write(&expiry)
    .build()?;

let hash = Sha256::digest(format!("{name}:{address}:{secret}").as_bytes());

assert_eq!(receipt.journal.bytes[..32], hash[..], name, address);
```

Zero Knowledge proof in Blockchain

Hash is stored in blockchain.

```rust
fn main() {
    let (name, address, secret, expiry): (String, String, String, u64) = env::read();
    let commitment = Sha256::digest(format!("{name}:{address}:{secret}").as_bytes());

    env::commit(&commitment); // Privacy-preserving
    env::commit(&expiry);
}
```

```rust
use risc0_zkvm::{default_prover, ExecutorEnv};
use sha2::{Digest, Sha256};

let name = "Alice".to_string();
let address = "123 Main St".to_string();
let expiry = 1690000000u64; // Timestamp
let secret = "secret-password"
```

```

let env = ExecutorEnv::builder()
    .write(&name)
    .write(&address)
    .write(&secret)
    .write(&expiry)
    .build()?;

let hash = Sha256::digest(format!("{name}:{address}:{secret}").as_bytes());

assert_eq!(receipt.journal.bytes[..32], hash[..]);
```

Rust

```
#[pallet::storage]
pub type Commitments<T: Config> = StorageMap<_, Blake2_128Concat, [u8; 32], (T::AccountId, u64)>;
// Maps (commitment) => (owner, expiry)
```

Rust

```
#[pallet::call]
impl<T: Config> Pallet<T> {
    #[pallet::weight(10_000)]
    pub fn register_kyc(
        origin: OriginFor<T>,
        proof: Vec<u8>,
        hash: [u8; 32],
        expiry: u64
    ) -> DispatchResult {
        let who = ensure_signed(origin)?;

        // Verify Risc0 proof
        risc0::verify_proof(&proof, &hash, &expiry)?;

        ensure!(
            !KycHashes::<T>::contains_key(&hash),
            Error::<T>::KycAlreadyRegistered
        );

        KycHashes::<T>::insert(&hash, (who.clone(), expiry));
        Ok(())
    }

    #[pallet::weight(10_000)]
    pub fn extend_kyc(
        origin: OriginFor<T>,
        proof: Vec<u8>,
        hash: [u8; 32],
        new_expiry: u64
    ) -> DispatchResult {
```

```
        let who = ensure_signed(origin)?;

        risc0::verify_proof(&proof, &hash, &new_expiry)?;

        KycHashes::<T>::try_mutate(&hash, |entry| {
            let (account, expiry) =
            entry.as_mut().ok_or(Error::<T>::NotRegistered)?;
            ensure!(*account == who, Error::<T>::NotAuthorized);
            *expiry = new_expiry;
            Ok(())
        })
    }

    #[pallet::weight(10_000)]
    pub fn governance_remove(
        origin: OriginFor<T>,
        hash: [u8; 32]
    ) -> DispatchResult {
        T::GovernanceOrigin::ensure_origin(origin)?;
        KycHashes::<T>::remove(&hash);
        Ok(())
    }
}
```

Store the name, address and photo by validator

```rust
/// Struct to hold identity information
#[derive(Encode, Decode, Clone, PartialEq, Eq, RuntimeDebug, TypeInfo, MaxEncodedLen)]
pub struct IdentityData {
    pub name: BoundedVec<u8, ConstU32<100>>,
    pub address: BoundedVec<u8, ConstU32<200>>,
    pub photo_hash: BoundedVec<u8, ConstU32<200>>,
    pub validator_approved: bool,
}

// Stored and approved by validator
#[pallet::storage]
#[pallet::getter(fn identity_data)]
pub type IdentityStore<T: Config> = StorageValue<_, IndentityData, ValueQuery>;
```

Users must ensure that their name, address, and photo are added to the blockchain by a validator, or they can appeal to the governance.