Decentralized Medical Records DApp Using Ethereum Smart Contracts

Authors: Carlos Eduardo Menezes - 2023252, Renan de Castilhos da Silva – 2023211
Module: *Distributed Digital Transactions*
GitHub Repo: *https://github.com/CaduOStudent/CA1-DDT-DAPP*

Context & Problem

This assignment has the objective of testing our knowledge in blockchain and smart contracts development by searching about blockchain, encryption techniques and business model along with the development of a decentralized app with smart contract for a specific business model.

For our assignment we've decided to develop a smart contract for a healthcare system and medical records. Where users could create medical records and the records would be assigned to their own address in the blockchain while they would become owners of these records. As owners they have the option of grant or revoke access to other users passing their records Id and the user address in the blockchain. Using the right modifiers, the only users(addresses) allowed to read the medical record data would be the owner and whoever has access to it.

For the inputs the user has to use the cid of the data it wants to create by using IPFS encryption along with the SHA256 hash for the same file. When another user has access to the file through the grant access function from the contract, it can have access to the cid and hash of the record and then can decrypt and access the file.

We also decided to create a function for other addresses to request access to records using the id of the file and when doing it, the owner would be notified.

The assignment also requested us to discuss about the benefits and challenges of applying blockchain to this specific use case and identify the most suitable cryptographic method for securing information in this scenario explaining how it enhances security in the blockchain

System Design & Architecture

Principle 1 Design Before Code

Goal: Minimal on-chain registry for medical-record **metadata** only; encrypted medical data stays off-chain (IPFS or other).

- **Store**: `cid`, `contentHash`, `owner`, `createdAt`.
- **Patient control**: per-record boolean read access granted/revoked to addresses.
- **Events**: `AccessRequested`, `AccessGranted`, `AccessRevoked`, `RecordCreated` drive off-chain consent UI, audit, and workflows.
- **Privacy**: contract never holds plaintext or decryption keys.

Participants and Roles

-**Patient (owner)**: creates records, always implicitly authorized, grants/revokes access.
-**Requester (clinician/researcher)**: emits access requests; reads metadata; off-chain services release decrypted content only after verifying on-chain permission and event history.
-**Auditor/Anyone**: reads public metadata and listens to events for compliance and logs.

Testing addresses:
-**ACCOUNT 1**: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
-**ACCOUNT 2**: 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2
-**ACCOUNT 3**: 0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db

Scope and Non-scope
In scope
- Patient-only `createRecord` (metadata minting).
- Event-driven off-chain request lifecycle (`AccessRequested`).
- Patient-managed `grantAccess` / `revokeAccess`.
- `canAccess` view and public metadata read.
- Minimal on-chain storage (metadata only).

Out of scope
- Storing plaintext medical data on-chain.
- Attribute-level permissions (only boolean per address).
- Role-based governance, emergency overrides, delegated revocation.
- Complex consent (time-bound, purpose-limited) unless extended off-chain.
- On-chain payments, billing, KYC.
- Immutability of off-chain content (events require off-chain correlation).

System Design & Architecture

Data Model and Access State Machine
Data model
-**records**: `mapping(uint256 => Record)` where `Record { owner: address; cid: string; contentHash: bytes32; createdAt: uint256 }`.
-**accessGranted**: `mapping(uint256 => mapping(address => bool))`.
-**nextRecordId**: `uint256` auto-increment.
-**Key invariant**: `records[id].owner != address(0)` ⇒ record exists.
States per record per address
-**NoAccess**: `accessGranted[id][addr] == false` and `addr != owner`.
-**Requested**: represented by `AccessRequested` event (on-chain mapping unchanged).
-**Granted**: `accessGranted[id][addr] == true`.
-**Revoked**: `accessGranted[id][addr] == false` after revoke (semantically distinct from NoAccess).
Transitions
- requestAccess → emits `AccessRequested`.
- grantAccess (owner only) → sets true, emits `AccessGranted`.
- revokeAccess (owner only) → sets false, emits `AccessRevoked`.
- Owner implicit access: owner always passes `canAccess`.
Transactions Views and Events
Transactions
- `createRecord(cid, contentHash) external returns (uint256)` — mints metadata; emits `RecordCreated`.
- `requestAccess(uint256 id) external` — emits `AccessRequested`.
- `grantAccess(uint256 id, address grantee) external onlyOwner(id)` — sets flag true; emits `AccessGranted`.
- `revokeAccess(uint256 id, address grantee) external onlyOwner(id)` — sets flag false; emits `AccessRevoked`.
Views
- `canAccess(uint256 id, address user) public view returns (bool)` — true if owner or accessGranted.
- `records(uint256) public view` — returns `Record` metadata.
Events (indexed fields recommended)
- `RecordCreated(recordId, owner, cid, contentHash)` — index `recordId`, `owner`.
- `AccessRequested(recordId, requester)` — index `recordId`, `requester`.
- `AccessGranted(recordId, grantee, by)` — index `recordId`, `grantee`, `by`.
- `AccessRevoked(recordId, grantee, by)` — index `recordId`, `grantee`, `by`.

Implementation & Testing

Our system uses the following architecture components:

- Smart Contract: contracts/MedicalRecords.sol, deployed on a local Ethereum network (Hardhat node or Ganache).
- Backend tooling: Hardhat for compilation, deployment (scripts/deploy.js), interaction (scripts/interact-example.js) and automated tests (test/medical.test.js).
- Wallet: MetaMask for patient and provider accounts and transaction signing.
- IDE: Remix for contract editing, compilation and manual testing using Injected Web3.
- Frontend: frontend/index.html and app.js using ethers.js to connect to the deployed contract and interact via MetaMask.

Architecture flow (diagram suggestion):

Patient (MetaMask) → Frontend / Remix → MedicalRecords smart contract → access permissions & events

Off-chain encrypted records are stored in IPFS or a database and referenced by cid and contentHash.

Cryptography & Security

The DApp combines on-chain Ethereum security with off-chain Python cryptography using SHA-256.

On-chain, we do not implement cryptographic algorithms directly in Solidity. Instead, we rely on Ethereum's ECDSA signatures, generated by MetaMask whenever the user sends a transaction. Ethereum nodes verify these ECDSA signatures and derive the sender address as `msg.sender`. In our contract, each `Record` has an `owner` field, and the `onlyOwner` modifier checks `records[id].owner == msg.sender`. Therefore, only the account that originally created the record (identified via ECDSA) can call `grantAccess` or `revokeAccess`. ECDSA does not appear in the Solidity code, but it is what guarantees the correctness of `msg.sender` and our access-control logic.

Off-chain, we use Python scripts to protect the actual medical data. The medical file is encrypted (e.g. with AES) and stored outside the blockchain (IPFS or a secure database). Python then computes a SHA-256 hash of the encrypted file and sends this hash, converted to `bytes32`, to the smart contract as `contentHash` in `createRecord`. Each `Record` stores the patient address, the `cid` (encrypted URI), the SHA-256 `contentHash`, and a timestamp.

Later, the same SHA-256 hash can be recomputed in Python from the off-chain file and compared with the on-chain `contentHash`. Any mismatch reveals tampering. By keeping only metadata and SHA-256 hashes on-chain (while delegating heavy cryptography to Python and signature verification to Ethereum/ECDSA), the solution provides integrity, access control and auditability, without ever exposing raw medical data on the blockchain.

```python
# First, let's create the missing key_utils module functionality
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import serialization, hashes

# This function would normally be in key_utils.py
def load_private_key(pem_data):
    private_key = serialization.load_pem_private_key(
        pem_data,
        password=None,
    )
    return private_key

# Now the rest of your code
import base64
import ipfshttpclient
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import hashes

def download_from_ipfs(cid: str, ipfs_addr="/dns/localhost/tcp/5001/http"):
    client = ipfshttpclient.connect(ipfs_addr)
    data = client.cat(cid)
    return data

def decrypt_sym_key(enc_key_b64: str, private_key):
    enc_bytes = base64.b64decode(enc_key_b64)
    sym_key = private_key.decrypt(
        enc_bytes,
        padding.OAEP(mgf=padding.MGF1(algorithm=hashes.SHA256()), algorithm=hashes.SHA256(), label=None)
    )
    return sym_key
def decrypt_file_from_package(package: dict, recipient_priv_pem: bytes, out_path: str, ipfs_addr="/dns/localhost/tcp/5001/http"):
    priv = load_private_key(recipient_priv_pem)
    cid = package["cid"]
    nonce = base64.b64decode(package["nonce"])
    enc_keys = package["enc_keys"]

    # Find recipient id if you store mapping; here we assume single recipient and pick their encrypted key
    # In practice, package.enc_keys keyed by recipient id; you must pick the correct one.
    # For demo, take first enc_key
    enc_key_b64 = next(iter(enc_keys.values()))
    sym_key = decrypt_sym_key(enc_key_b64, priv)

    ciphertext = download_from_ipfs(cid, ipfs_addr)
    aesgcm = AESGCM(sym_key)
    plaintext = aesgcm.decrypt(nonce, ciphertext, None)
    with open(out_path, "wb") as f:
        f.write(plaintext)
    return out_path

# Example usage (not for production)
#if __name__ == "__main__":
#    import json
#    pkg = json.load(open("package.json"))
#    priv_pem = open("alice_priv.pem","rb").read()
#    decrypt_file_from_package(pkg, priv_pem, "recovered_scan.pdf")
```

Proposed Blockchain Solution and Business Model

Smart Contract (MedicalRecords):
Acts as a trusted layer between patients and healthcare providers.

Stores only metadata:

cid: IPFS CID / encrypted URI to the off-chain record
contentHash: integrity hash
owner: patient's address

Key functions:

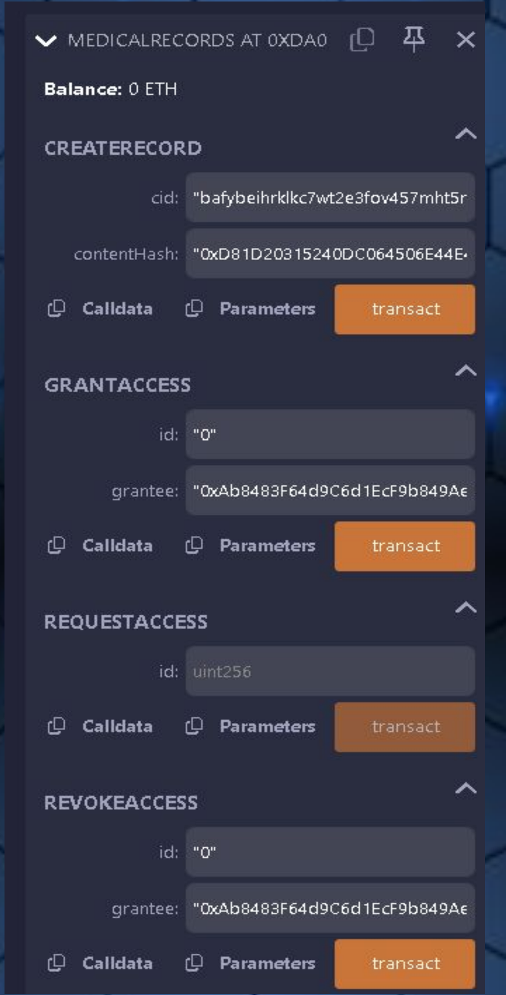
createRecord(): patient registers a newrecord
requestAccess(): provider asks for access
grantAccess() / revokeAccess(): patient manages permissions
canAccess(): checks if a user is authorized

```
Algorithm      Hash
---------      ----
SHA256         5CC1C245CA1AE3C4C40959ED255CFC031E1E2E283FCCD05B74F68510E7090208
```

NAME	CID
Tutorial 6 - 2025 (1).pdf	bafyb...3vyl4

Contract code

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;

contract MedicalRecords {
    //address public owner;
    struct Record {
        address owner;
        string cid;        // IPFS CID or encrypted URI
        bytes32 contentHash; // keccak256 or SHA-256 represented as bytes32
        uint256 createdAt;
    }

    //constructor() {
    //owner = msg.sender; // Deployer is the owner
//}

    mapping(uint256 => Record) public records;
    mapping(uint256 => mapping(address => bool)) public accessGranted;
    uint256 public nextRecordId;

    event RecordCreated(uint256 indexed recordId, address indexed owner, string
cid, bytes32 contentHash);
    event AccessRequested(uint256 indexed recordId, address indexed requester);
    event AccessGranted(uint256 indexed recordId, address indexed grantee,
address indexed by);
    event AccessRevoked(uint256 indexed recordId, address indexed grantee,
address indexed by);

    modifier onlyOwner(uint256 id) {
        require(records[id].owner == msg.sender, "Not owner");
        _;
    }

    function createRecord(string calldata cid, bytes32 contentHash) external
returns (uint256) {
        uint256 id = nextRecordId++;
        records[id] = Record({ owner: msg.sender, cid: cid, contentHash:
contentHash, createdAt: block.timestamp });
        emit RecordCreated(id, msg.sender, cid, contentHash);
        return id;
    }

    function requestAccess(uint256 id) external {
        require(records[id].owner != address(0), "No such record");
        emit AccessRequested(id, msg.sender);
    }

    function grantAccess(uint256 id, address grantee) external onlyOwner(id) {
        accessGranted[id][grantee] = true;
        emit AccessGranted(id, grantee, msg.sender);
    }

    function revokeAccess(uint256 id, address grantee) external onlyOwner(id) {
        accessGranted[id][grantee] = false;
        emit AccessRevoked(id, grantee, msg.sender);
    }

    function canAccess(uint256 id, address user) public view returns (bool) {
        if (records[id].owner == user) return true;
        return accessGranted[id][user];
    }

  //  function getOwner() public view returns (address) {
   // return owner;
//}
}
```



Benefits and Challenges

Benefits

- Patient-centric control over medical record access.
- Immutable, transparent log of consent and access requests.
- Easier cross-institution data sharing via a shared consent layer.
- Strong integrity and non-repudiation through digital signatures and hashes.

Challenges

- Mapping blockchain addresses to real-world identities under GDPR and healthcare regulations.
- Key management and recovery for non-technical users using MetaMask.
- Gas costs and scalability on public networks; sidechains or Layer-2 solutions may be required.
- Ensuring secure, encrypted off-chain storage and correct linkage between CIDs and hashes.