LLM-generated Solidity Smart Contracts using Arena Chatbot

Prompt

Generate a Solidity smart contract for a controlled medication administration system. The contract should include role-based access control to manage different stakeholder roles. Implement functions to track the production, distribution, prescription, administration, and disposal of medication batches. Ensure that all actions emit events for traceability, and incorporate approval mechanisms to comply with regulatory requirements.

Response 1: GPT-4o

Below is a Solidity smart contract for a controlled medication administration system. It includes role-based access control, the ability to track medication production, distribution, prescription, administration, and disposal, and emits events for traceability. It also incorporates approval mechanisms.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/access/AccessControl.sol";

contract ControlledMedicationSystem is AccessControl {
    // Define roles
    bytes32 public constant PRODUCER_ROLE = keccak256("PRODUCER_ROLE");
    bytes32 public constant DISTRIBUTOR_ROLE =
keccak256("DISTRIBUTOR_ROLE");
    bytes32 public constant PRESCRIBER_ROLE =
keccak256("PRESCRIBER_ROLE");
    bytes32 public constant ADMINISTRATOR_ROLE
=keccak256("ADMINISTRATOR_ROLE");
    bytes32 public constant REGULATOR_ROLE = keccak256("REGULATOR_ROLE");

    // Struct to represent a medication batch
    struct MedicationBatch {
        uint256 batchId;
        string name;
        uint256 productionDate;
        uint256 expirationDate;
        address producer;
        bool approvedByRegulator;
    }

    // Struct to represent a medication prescription
    struct Prescription {
```

```
        uint256 prescriptionId;
        uint256 batchId;
        address prescriber;
        address patient;
        uint256 quantity;
        bool filled;
    }

    // State variables
    uint256 public nextBatchId = 1;
    uint256 public nextPrescriptionId = 1;
    mapping(uint256 => MedicationBatch) public medicationBatches;
    mapping(uint256 => Prescription) public prescriptions;

    // Events for traceability
    event MedicationProduced(uint256 batchId, string name, uint256
productionDate, uint256 expirationDate, address producer);
    event MedicationApproved(uint256 batchId, address regulator);
    event MedicationDistributed(uint256 batchId, address distributor);
    event MedicationPrescribed(uint256 prescriptionId, uint256 batchId,
address prescriber, address patient, uint256 quantity);
    event MedicationAdministered(uint256 prescriptionId, address
administrator, address patient, uint256 quantity);
    event MedicationDisposed(uint256 batchId, address disposer);

    // Modifier to ensure batch exists
    modifier batchExists(uint256 batchId) {
        require(medicationBatches[batchId].batchId != 0, "Batch does not
exist");
        _;
    }

    // Constructor
    constructor() {
        _setupRole(DEFAULT_ADMIN_ROLE, msg.sender);
    }

    // Function to produce medication
    function produceMedication(
        string memory name,
        uint256 productionDate,
        uint256 expirationDate
    ) public onlyRole(PRODUCER_ROLE) {
        uint256 batchId = nextBatchId++;
        medicationBatches[batchId] = MedicationBatch({
```

```
            batchId: batchId,
            name: name,
            productionDate: productionDate,
            expirationDate: expirationDate,
            producer: msg.sender,
            approvedByRegulator: false
        });

        emit MedicationProduced(batchId, name, productionDate,
expirationDate, msg.sender);
    }

    // Function for regulator to approve a medication batch
    function approveMedication(uint256 batchId) public
onlyRole(REGULATOR_ROLE) batchExists(batchId) {
        MedicationBatch storage batch = medicationBatches[batchId];
        require(!batch.approvedByRegulator, "Batch already approved");
        batch.approvedByRegulator = true;

        emit MedicationApproved(batchId, msg.sender);
    }

    // Function to distribute medication
    function distributeMedication(uint256 batchId) public
onlyRole(DISTRIBUTOR_ROLE) batchExists(batchId) {
        MedicationBatch storage batch = medicationBatches[batchId];
        require(batch.approvedByRegulator, "Batch not approved by
regulator");

        emit MedicationDistributed(batchId, msg.sender);
    }

    // Function to prescribe medication
    function prescribeMedication(
        uint256 batchId,
        address patient,
        uint256 quantity
    ) public onlyRole(PRESCRIBER_ROLE) batchExists(batchId) {
        MedicationBatch storage batch = medicationBatches[batchId];
        require(batch.approvedByRegulator, "Batch not approved by
regulator");

        uint256 prescriptionId = nextPrescriptionId++;
        prescriptions[prescriptionId] = Prescription({
            prescriptionId: prescriptionId,
```

```
            batchId: batchId,
            prescriber: msg.sender,
            patient: patient,
            quantity: quantity,
            filled: false
        });

        emit MedicationPrescribed(prescriptionId, batchId, msg.sender,
patient, quantity);
    }

    // Function to administer medication
    function administerMedication(
        uint256 prescriptionId
    ) public onlyRole(ADMINISTRATOR_ROLE) {
        Prescription storage prescription = prescriptions[prescriptionId];
        require(!prescription.filled, "Prescription already filled");

        prescription.filled = true;

        emit MedicationAdministered(prescriptionId, msg.sender,
prescription.patient, prescription.quantity);
    }

    // Function to dispose of medication
    function disposeMedication(uint256 batchId) public
onlyRole(PRODUCER_ROLE) batchExists(batchId) {
        delete medicationBatches[batchId];

        emit MedicationDisposed(batchId, msg.sender);
    }
}
```

Response 2: Claude 3.5 Sonnet

Here's a comprehensive Solidity smart contract for a controlled medication administration system with role-based access control and traceability features:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/access/AccessControl.sol";
import "@openzeppelin/contracts/security/Pausable.sol";
import "@openzeppelin/contracts/utils/Counters.sol";

contract MedicationControl is AccessControl, Pausable {
    using Counters for Counters.Counter;

    // Role definitions
    bytes32 public constant MANUFACTURER_ROLE =
keccak256("MANUFACTURER_ROLE");
    bytes32 public constant DISTRIBUTOR_ROLE =
keccak256("DISTRIBUTOR_ROLE");
    bytes32 public constant PHYSICIAN_ROLE = keccak256("PHYSICIAN_ROLE");
    bytes32 public constant PHARMACIST_ROLE =
keccak256("PHARMACIST_ROLE");
    bytes32 public constant REGULATOR_ROLE = keccak256("REGULATOR_ROLE");

    Counters.Counter private _batchIds;
    Counters.Counter private _prescriptionIds;

    // Structs
    struct MedicationBatch {
        uint256 batchId;
        string medicationName;
        uint256 quantity;
        address manufacturer;
        BatchStatus status;
        uint256 manufactureDate;
        uint256 expiryDate;
        string ipfsHash; // Additional documentation stored on IPFS
    }

    struct Prescription {
        uint256 prescriptionId;
        uint256 batchId;
        address patient;
        address physician;
        uint256 quantity;
```

```
        uint256 issueDate;
        uint256 validUntil;
        PrescriptionStatus status;
    }

    enum BatchStatus {
        Manufactured,
        QualityChecked,
        Distributed,
        Dispensed,
        Disposed
    }

    enum PrescriptionStatus {
        Issued,
        Approved,
        Dispensed,
        Completed,
        Cancelled
    }

    // Mappings
    mapping(uint256 => MedicationBatch) public medicationBatches;
    mapping(uint256 => Prescription) public prescriptions;
    mapping(uint256 => mapping(address => bool)) public batchApprovals;

    // Events
    event BatchManufactured(uint256 batchId, string medicationName,
address manufacturer);
    event BatchQualityChecked(uint256 batchId, address inspector);
    event BatchDistributed(uint256 batchId, address distributor);
    event BatchDispensed(uint256 batchId, address pharmacist);
    event BatchDisposed(uint256 batchId, address disposer);
    event PrescriptionIssued(uint256 prescriptionId, address patient,
address physician);
    event PrescriptionDispensed(uint256 prescriptionId, address
pharmacist);

    constructor() {
        _setupRole(DEFAULT_ADMIN_ROLE, msg.sender);
    }

    // Modifiers
    modifier onlyManufacturer() {
```

```
        require(hasRole(MANUFACTURER_ROLE, msg.sender), "Caller is not a
manufacturer");
        _;
    }

    modifier onlyPhysician() {
        require(hasRole(PHYSICIAN_ROLE, msg.sender), "Caller is not a
physician");
        _;
    }

    modifier onlyPharmacist() {
        require(hasRole(PHARMACIST_ROLE, msg.sender), "Caller is not a
pharmacist");
        _;
    }

    modifier onlyRegulator() {
        require(hasRole(REGULATOR_ROLE, msg.sender), "Caller is not a
regulator");
        _;
    }

    // Manufacturing functions
    function manufactureBatch(
        string memory medicationName,
        uint256 quantity,
        uint256 expiryDate,
        string memory ipfsHash
    ) public onlyManufacturer whenNotPaused returns (uint256) {
        _batchIds.increment();
        uint256 newBatchId = _batchIds.current();

        medicationBatches[newBatchId] = MedicationBatch({
            batchId: newBatchId,
            medicationName: medicationName,
            quantity: quantity,
            manufacturer: msg.sender,
            status: BatchStatus.Manufactured,
            manufactureDate: block.timestamp,
            expiryDate: expiryDate,
            ipfsHash: ipfsHash
        });

        emit BatchManufactured(newBatchId, medicationName, msg.sender);
```

```
        return newBatchId;
    }

    // Quality control functions
    function approveQualityCheck(uint256 batchId) public onlyRegulator
whenNotPaused {
        require(medicationBatches[batchId].status ==
BatchStatus.Manufactured, "Invalid batch status");
        medicationBatches[batchId].status = BatchStatus.QualityChecked;
        emit BatchQualityChecked(batchId, msg.sender);
    }

    // Distribution functions
    function distributeBatch(uint256 batchId) public whenNotPaused {
        require(hasRole(DISTRIBUTOR_ROLE, msg.sender), "Caller is not a
distributor");
        require(medicationBatches[batchId].status ==
BatchStatus.QualityChecked, "Batch not quality checked");

        medicationBatches[batchId].status = BatchStatus.Distributed;
        emit BatchDistributed(batchId, msg.sender);
    }

    // Prescription functions
    function issuePrescription(
        uint256 batchId,
        address patient,
        uint256 quantity,
        uint256 validityDays
    ) public onlyPhysician whenNotPaused returns (uint256) {
        require(medicationBatches[batchId].status ==
BatchStatus.Distributed, "Medication not available");

        _prescriptionIds.increment();
        uint256 newPrescriptionId = _prescriptionIds.current();

        prescriptions[newPrescriptionId] = Prescription({
            prescriptionId: newPrescriptionId,
            batchId: batchId,
            patient: patient,
            physician: msg.sender,
            quantity: quantity,
            issueDate: block.timestamp,
            validUntil: block.timestamp + (validityDays * 1 days),
            status: PrescriptionStatus.Issued
```

```
        });

        emit PrescriptionIssued(newPrescriptionId, patient, msg.sender);
        return newPrescriptionId;
    }

    // Dispensing functions
    function dispensePrescription(uint256 prescriptionId) public
onlyPharmacist whenNotPaused {
        Prescription storage prescription = prescriptions[prescriptionId];
        require(prescription.status == PrescriptionStatus.Issued, "Invalid
prescription status");
        require(block.timestamp <= prescription.validUntil, "Prescription
expired");

        MedicationBatch storage batch =
medicationBatches[prescription.batchId];
        require(batch.quantity >= prescription.quantity, "Insufficient
medication quantity");

        batch.quantity -= prescription.quantity;
        prescription.status = PrescriptionStatus.Dispensed;

        emit PrescriptionDispensed(prescriptionId, msg.sender);
    }

    // Disposal functions
    function disposeBatch(uint256 batchId) public whenNotPaused {
        require(hasRole(MANUFACTURER_ROLE, msg.sender) ||
hasRole(PHARMACIST_ROLE, msg.sender),
                "Unauthorized to dispose");
        require(medicationBatches[batchId].status != BatchStatus.Disposed,
"Already disposed");

        medicationBatches[batchId].status = BatchStatus.Disposed;
        emit BatchDisposed(batchId, msg.sender);
    }

    // Administrative functions
    function pause() public onlyRole(DEFAULT_ADMIN_ROLE) {
        _pause();
    }

    function unpause() public onlyRole(DEFAULT_ADMIN_ROLE) {
        _unpause();
```

```
    }

    // Getter functions
    function getBatchDetails(uint256 batchId) public view returns
(MedicationBatch memory) {
        return medicationBatches[batchId];
    }

    function getPrescriptionDetails(uint256 prescriptionId) public view
returns (Prescription memory) {
        return prescriptions[prescriptionId];
    }
}
```

Response 3: Gemini 2.0 Flash

Okay, let's craft a Solidity smart contract for a controlled medication administration system with role-based access control, tracking, events, and approval mechanisms. This is a complex example, so we'll break it down and provide explanations along the way.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract ControlledMedicationSystem {

    // --- Enums ---
    enum MedicationStatus {
        PRODUCTION,
        DISTRIBUTED,
        PRESCRIBED,
        ADMINISTERED,
        DISPOSED
    }

    // --- Structs ---
    struct MedicationBatch {
        uint batchId;
        string name;
        uint productionDate;
        uint expiryDate; //Timestamp
        address manufacturer;
        address distributor;
        address prescriber;
        address administrator;
        address disposer;
        MedicationStatus status;
    }

    // --- State Variables ---
    mapping(uint => MedicationBatch) public medicationBatches;
    uint public batchCounter;


    // Role-Based Access Control
    mapping(address => bool) public isManufacturer;
    mapping(address => bool) public isDistributor;
    mapping(address => bool) public isPrescriber;
    mapping(address => bool) public isAdministrator;
    mapping(address => bool) public isDisposer;
    address public owner;
```

```
    // Approval Mechanism
    mapping(uint => bool) public distributionApproved;
    mapping(uint => bool) public prescriptionApproved;
    mapping(uint => bool) public administrationApproved;
    mapping(uint => bool) public disposalApproved;

    // Modifier for access control
    modifier onlyManufacturer() {
        require(isManufacturer[msg.sender], "Only manufacturers can
perform this action.");
        _;
    }

    modifier onlyDistributor() {
        require(isDistributor[msg.sender], "Only distributors can perform
this action.");
        _;
    }

    modifier onlyPrescriber() {
        require(isPrescriber[msg.sender], "Only prescribers can perform
this action.");
        _;
    }

    modifier onlyAdministrator() {
        require(isAdministrator[msg.sender], "Only administrators can
perform this action.");
        _;
    }

    modifier onlyDisposer() {
        require(isDisposer[msg.sender], "Only disposers can perform this
action.");
        _;
    }

    // --- Events ---
    event BatchProduced(uint batchId, string name, address manufacturer,
uint productionDate, uint expiryDate);
    event BatchDistributed(uint batchId, address distributor, uint
timestamp);
    event BatchPrescribed(uint batchId, address prescriber, uint
timestamp);
```

```
    event BatchAdministered(uint batchId, address administrator, uint
timestamp);
    event BatchDisposed(uint batchId, address disposer, uint timestamp);
    event RoleAssigned(address account, string role);
    event DistributionApprovalRequested(uint batchId, address
distributor);
    event PrescriptionApprovalRequested(uint batchId, address prescriber);
    event AdministrationApprovalRequested(uint batchId, address
administrator);
    event DisposalApprovalRequested(uint batchId, address disposer);
    event DistributionApproved(uint batchId, bool approved);
    event PrescriptionApproved(uint batchId, bool approved);
    event AdministrationApproved(uint batchId, bool approved);
    event DisposalApproved(uint batchId, bool approved);


    // --- Constructor ---
    constructor() {
        owner = msg.sender;
    }

    // --- Role Assignment Functions ---
    function addManufacturer(address _manufacturer) public  {
        require(msg.sender == owner, "Only owner can add a manufacturer");
        isManufacturer[_manufacturer] = true;
        emit RoleAssigned(_manufacturer, "Manufacturer");
    }

    function addDistributor(address _distributor) public {
        require(msg.sender == owner, "Only owner can add a distributor");
        isDistributor[_distributor] = true;
        emit RoleAssigned(_distributor, "Distributor");
    }

    function addPrescriber(address _prescriber) public {
        require(msg.sender == owner, "Only owner can add a prescriber");
        isPrescriber[_prescriber] = true;
        emit RoleAssigned(_prescriber, "Prescriber");
    }

    function addAdministrator(address _administrator) public {
        require(msg.sender == owner, "Only owner can add an
administrator");
        isAdministrator[_administrator] = true;
        emit RoleAssigned(_administrator, "Administrator");
```

```
    }

    function addDisposer(address _disposer) public {
        require(msg.sender == owner, "Only owner can add a disposer");
        isDisposer[_disposer] = true;
        emit RoleAssigned(_disposer, "Disposer");
    }

    // --- Medication Batch Management Functions ---
    function produceBatch(string memory _name, uint _expiryDate) public
onlyManufacturer {
        batchCounter++;
        medicationBatches[batchCounter] = MedicationBatch(
            batchCounter,
            _name,
            block.timestamp,
            _expiryDate,
            msg.sender,
            address(0),
            address(0),
            address(0),
            address(0),
            MedicationStatus.PRODUCTION
        );
        emit BatchProduced(batchCounter, _name, msg.sender,
block.timestamp, _expiryDate);
    }

    function distributeBatch(uint _batchId, address _distributor) public
onlyManufacturer {
        require(medicationBatches[_batchId].status ==
MedicationStatus.PRODUCTION, "Batch not in production status");
        medicationBatches[_batchId].distributor = _distributor;
        medicationBatches[_batchId].status = MedicationStatus.DISTRIBUTED;
        emit BatchDistributed(_batchId, _distributor, block.timestamp);
        emit DistributionApprovalRequested(_batchId, _distributor);
    }

    function approveDistribution(uint _batchId, bool _approved) public
onlyDistributor{
        distributionApproved[_batchId] = _approved;
        emit DistributionApproved(_batchId, _approved);
    }
```

```
    function prescribeBatch(uint _batchId, address _prescriber) public
onlyDistributor {
        require(medicationBatches[_batchId].status ==
MedicationStatus.DISTRIBUTED, "Batch not distributed yet");
        require(distributionApproved[_batchId], "Distribution not
approved");
        medicationBatches[_batchId].prescriber = _prescriber;
        medicationBatches[_batchId].status = MedicationStatus.PRESCRIBED;
         emit BatchPrescribed(_batchId, _prescriber, block.timestamp);
        emit PrescriptionApprovalRequested(_batchId, _prescriber);
    }

    function approvePrescription(uint _batchId, bool _approved) public
onlyPrescriber{
        prescriptionApproved[_batchId] = _approved;
        emit PrescriptionApproved(_batchId, _approved);
    }


    function administerBatch(uint _batchId, address _administrator) public
onlyPrescriber {
        require(medicationBatches[_batchId].status ==
MedicationStatus.PRESCRIBED, "Batch not prescribed yet");
        require(prescriptionApproved[_batchId], "Prescription not
approved");
        medicationBatches[_batchId].administrator = _administrator;
        medicationBatches[_batchId].status =
MedicationStatus.ADMINISTERED;
        emit BatchAdministered(_batchId, _administrator, block.timestamp);
        emit AdministrationApprovalRequested(_batchId, _administrator);
    }

    function approveAdministration(uint _batchId, bool _approved) public
onlyAdministrator{
        administrationApproved[_batchId] = _approved;
        emit AdministrationApproved(_batchId, _approved);
    }

     function disposeBatch(uint _batchId, address _disposer) public
onlyAdministrator {
        require(medicationBatches[_batchId].status ==
MedicationStatus.ADMINISTERED, "Batch not administered yet");
        require(administrationApproved[_batchId], "Administration not
approved");
        medicationBatches[_batchId].disposer = _disposer;
```

```
        medicationBatches[_batchId].status = MedicationStatus.DISPOSED;
        emit BatchDisposed(_batchId, _disposer, block.timestamp);
        emit DisposalApprovalRequested(_batchId, _disposer);
     }

     function approveDisposal(uint _batchId, bool _approved) public
onlyDisposer{
         disposalApproved[_batchId] = _approved;
         emit DisposalApproved(_batchId, _approved);
    }


    // --- Getter function for batch details ---
    function getBatchDetails(uint _batchId) public view returns
(MedicationBatch memory) {
        require(_batchId > 0 && _batchId <= batchCounter, "Invalid batch
ID");
        return medicationBatches[_batchId];
    }

    function getStatus(uint _batchId) public view
returns(MedicationStatus){
        return medicationBatches[_batchId].status;
    }
}
```