Documentation of Testing Tools for Blockchain Application

Team Members: *Sanjana Das, Anjali Bhatt, Subhra Mukherjee*

Summary: *This project implements two interoperable smart contracts: one for patients and one for doctors. The Patient Smart Contract stores medical records and manages access control, while the Doctor Smart Contract allows authorized doctors to view and update patient data. All updates are synced to ensure data consistency and secure sharing across the system.*

Main Problem Statement: Develop an interoperable system utilizing two distinct smart contracts: a Patient Smart Contract and a Doctor Smart Contract. The Patient Smart Contract will manage patient-provided medical records and control access permissions for healthcare providers(here Doctor). The Doctor Smart Contract will enable authorized doctors to retrieve patient records, leveraging the patient's address and a unique patient identifier. Subsequently, doctors can update medication details and clinical notes, with these modifications seamlessly reflected within the Patient Smart Contract, ensuring data consistency across both contracts.

1] Compiling and deploying the Patient smart contract

Problem statement:

Developing a Patient Smart contract that gives Patients exclusive control over entering, editing, and modifying their personal details.

Programming:

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

contract patientInfo

{

   struct MedicalRecord

   {
```

```
        uint256 patientId;
        string name;
        uint256 age;
        string bloodGroup;
        string allergies;
        string medications;
        string surgeries;
        string physicianNotes;
    }

    mapping(address => MedicalRecord[]) public medicalRecords;

    // Add a new medical record for the calling user
    function addMedicalRecord(
        uint256 _patientId,
        string memory _name,
        uint256 _age,
        string memory _bloodGroup,
        string memory _allergies,
        string memory _medications,
        string memory _surgeries,
        string memory _physicianNotes
    ) public
    {
        require(msg.sender == tx.origin);

        // Prevent duplicate record by name for this user
        for (uint256 i = 0; i < medicalRecords[msg.sender].length; i++)
        {
            if (keccak256(abi.encodePacked(medicalRecords[msg.sender][i].name)) ==
keccak256(abi.encodePacked(_name)))
```

```
        {
            return;
        }
    }

    medicalRecords[msg.sender].push(
        MedicalRecord(
            _patientId,
            _name,
            _age,
            _bloodGroup,
            _allergies,
            _medications,
            _surgeries,
            _physicianNotes
        )
    );
}

// Update an existing record by patientId for the calling user
// Only non-default (non-empty or non-zero) values will be updated
function updateMedicalRecord(
    uint256 _patientId,
    string memory _name,
    uint256 _age,
    string memory _bloodGroup,
    string memory _allergies,
    string memory _medications,
    string memory _surgeries,
    string memory _physicianNotes
```

```
) public
{
    require(msg.sender == tx.origin, "Only direct user accounts allowed");

    MedicalRecord[] storage records = medicalRecords[msg.sender];
    bool found = false;

    for (uint256 i = 0; i < records.length; i++)
    {
        if (records[i].patientId == _patientId)
        {
            // Update only if new value is provided

            if (bytes(_name).length > 0)
            {
                records[i].name = _name;
            }
            if (_age != 0)
            {
                records[i].age = _age;
            }
            if (bytes(_bloodGroup).length > 0)
            {
                records[i].bloodGroup = _bloodGroup;
            }
            if (bytes(_allergies).length > 0)
            {
                records[i].allergies = _allergies;
            }
            if (bytes(_medications).length > 0)
```

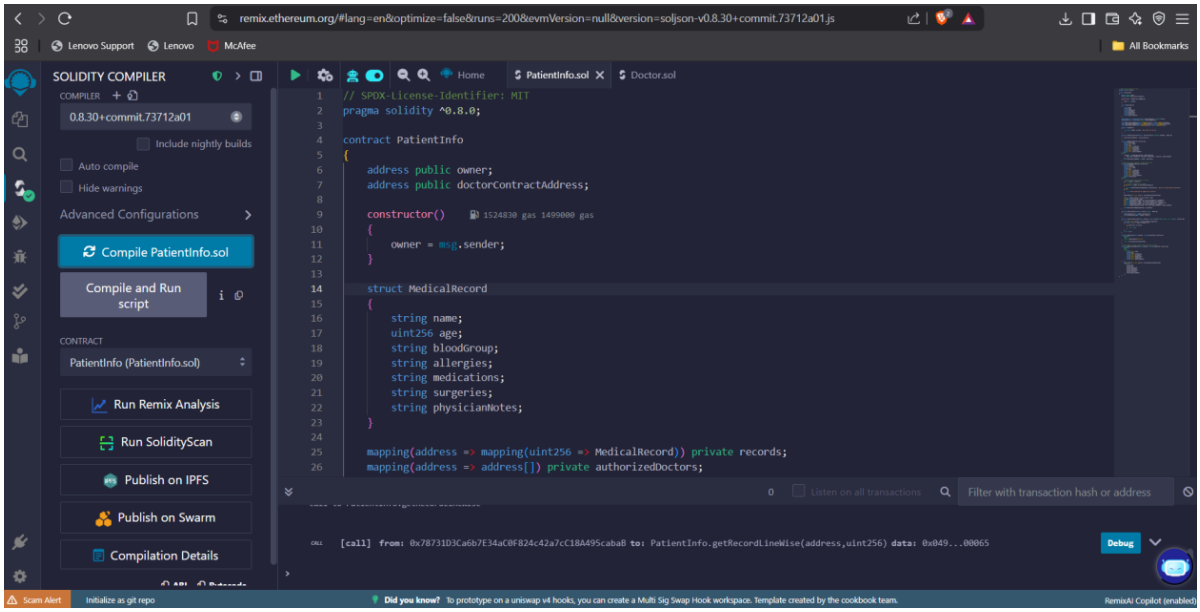
```
            {
                records[i].medications = _medications;
            }
            if (bytes(_surgeries).length > 0)
            {
                records[i].surgeries = _surgeries;
            }
            if (bytes(_physicianNotes).length > 0)
            {
                records[i].physicianNotes = _physicianNotes;
            }

            found = true;
            break;
        }
    }

    require(found, "Medical record not found for given patientId");
}

// Retrieve medical records matching patientID for the caller
function getMedicalRecord(uint256 patientID)
    public
    view
    returns (
        string[] memory name,
        uint256[] memory id,
        uint256[] memory age,
        string[] memory bloodGroup,
        string[] memory allergies,
```

```
        string[] memory medications,
        string[] memory surgeries,
        string[] memory physicianNotes
    )
{
    require(msg.sender == tx.origin);

    MedicalRecord[] storage records = medicalRecords[msg.sender];
    uint256 count = 0;

    // Count matching records
    for (uint256 i = 0; i < records.length; i++)
    {
        if (records[i].patientId == patientID)
        {
            count++;
        }
    }

    // Initialize arrays for results
    name = new string[](count);
    id = new uint256[](count);
    age = new uint256[](count);
    bloodGroup = new string[](count);
    allergies = new string[](count);
    medications = new string[](count);
    surgeries = new string[](count);
    physicianNotes = new string[](count);

    uint256 index = 0;
```

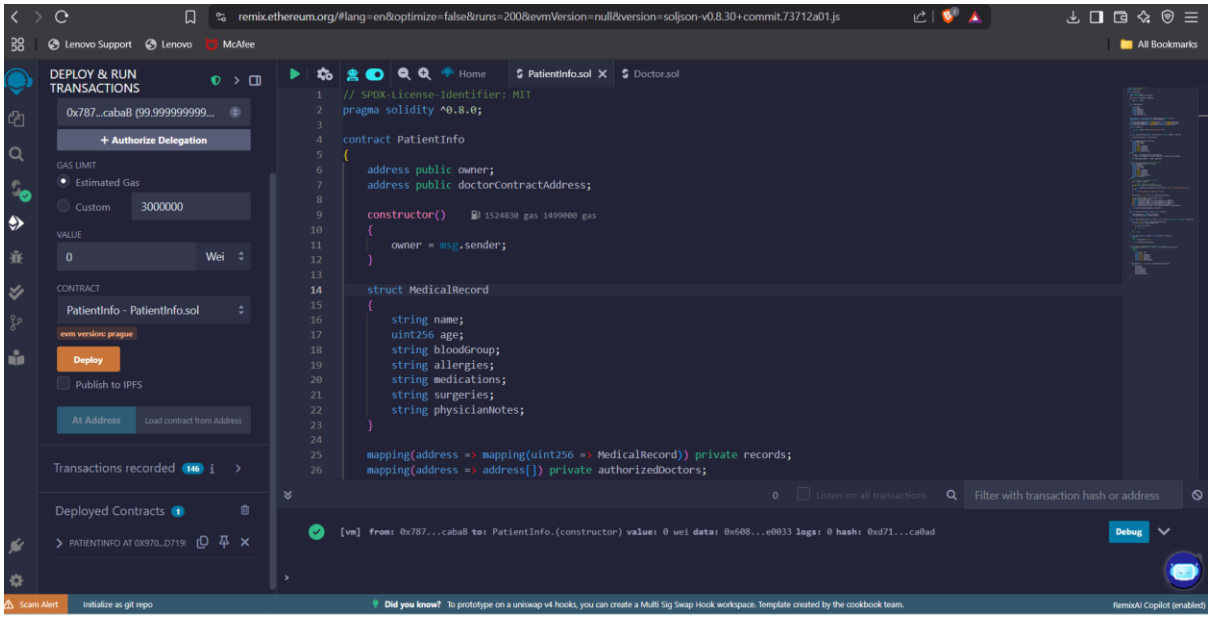
```
        // Populate arrays with matching records
        for (uint256 i = 0; i < records.length; i++)
        {
            if (records[i].patientId == patientID)
            {
                name[index] = records[i].name;
                id[index] = records[i].patientId;
                age[index] = records[i].age;
                bloodGroup[index] = records[i].bloodGroup;
                allergies[index] = records[i].allergies;
                medications[index] = records[i].medications;
                surgeries[index] = records[i].surgeries;
                physicianNotes[index] = records[i].physicianNotes;
                index++;
            }
        }
    }
}
```

Execution and deployment of the smart contract:

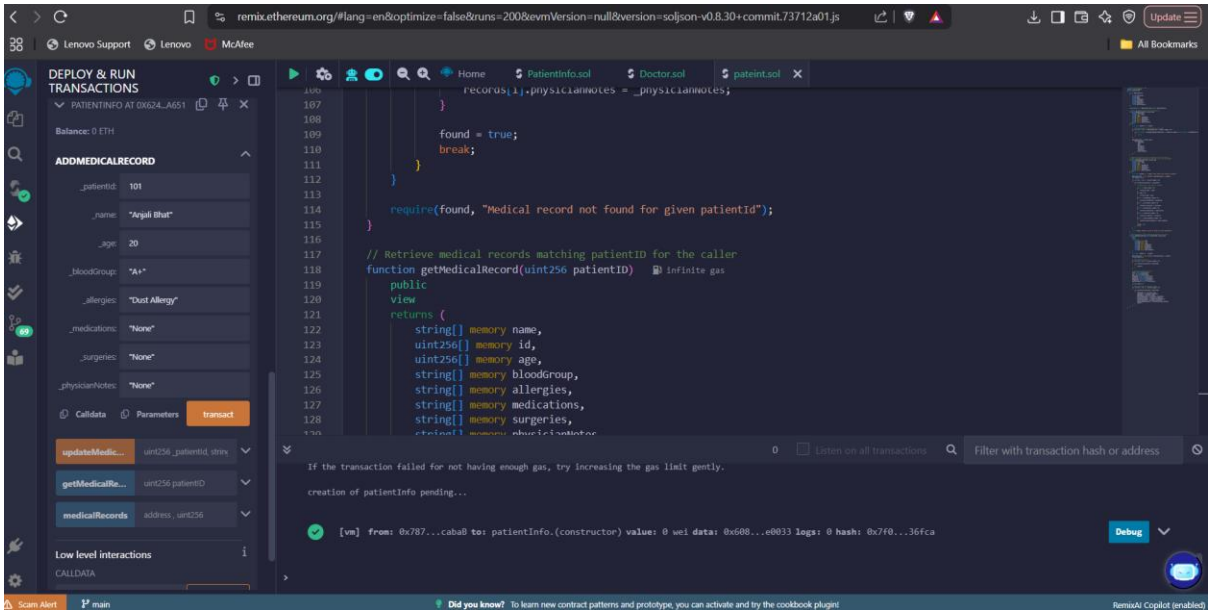
Step 1: Compilation of the Patient Smart Contract



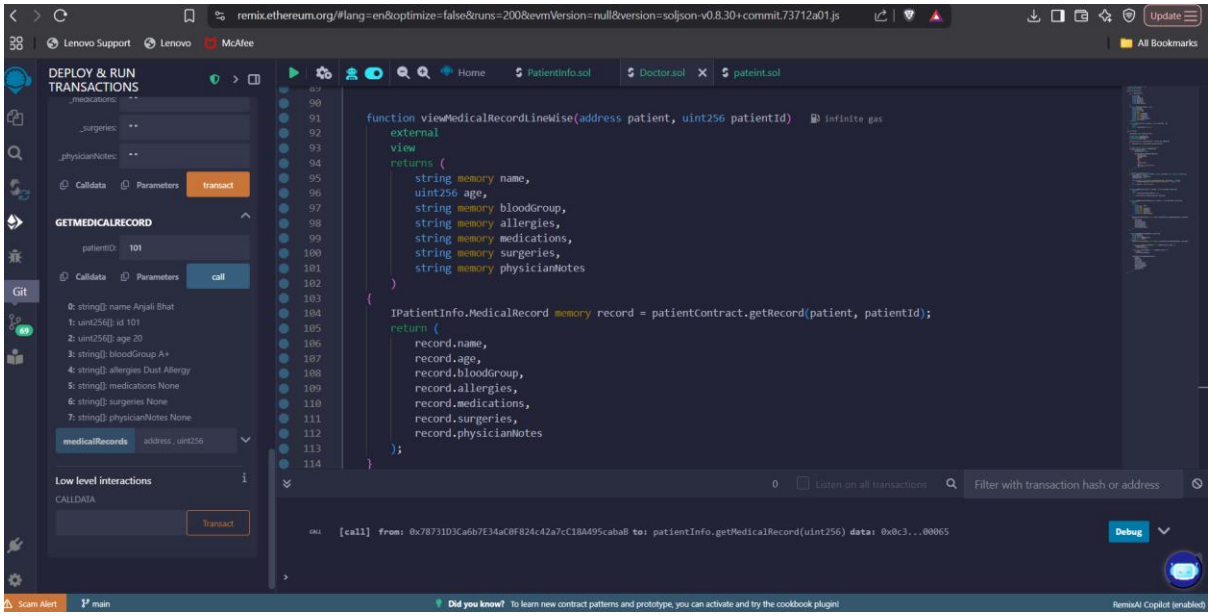
Step 2: Deploy the Patient smart contract using the first address



Step 3: Entering the patient's details and transacting it



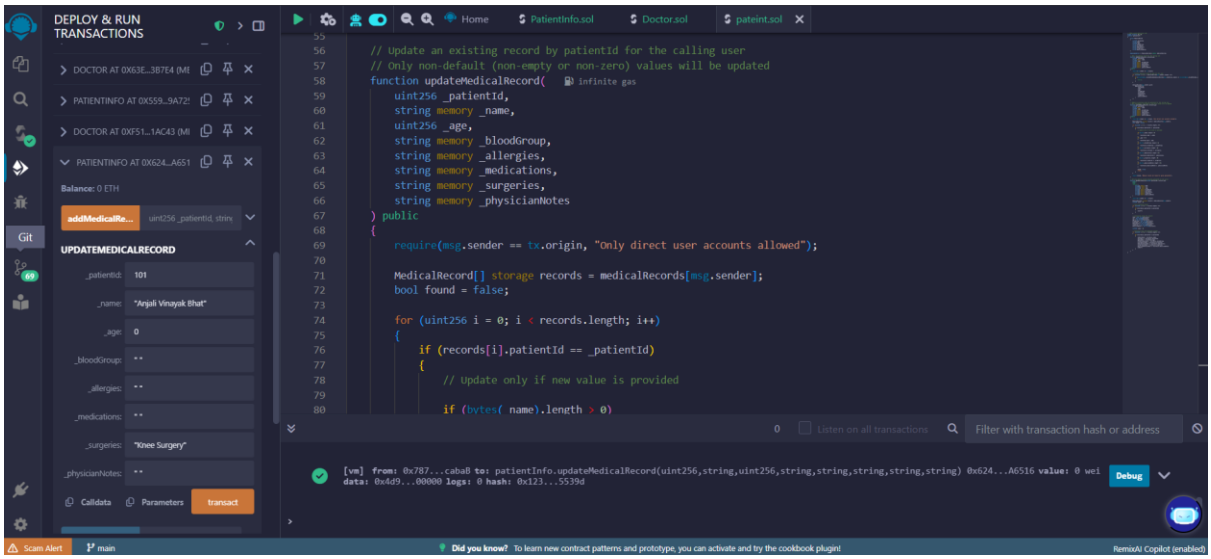
Now viewing the entered details of the patient.



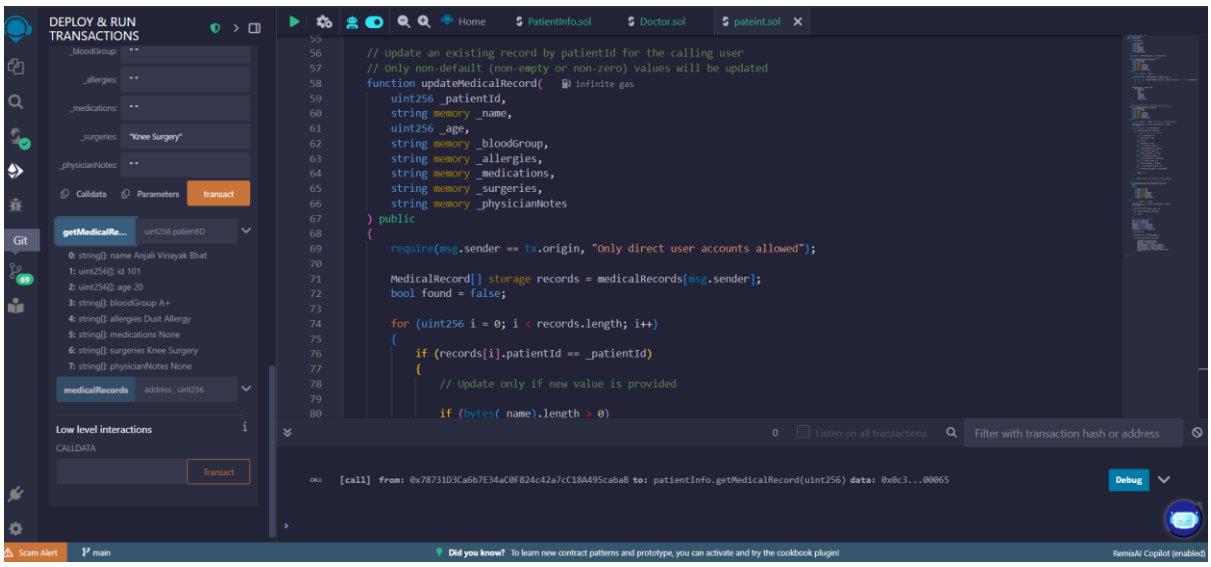
Step 4: Then update the details of the patient. *[note this step is only required if you want to update the details of the patient else you can skip this step]*

Here we only update the fields we want to and keep the rest as none to retrieve from the previous information.

In the image below updating, name of the patient and their surgery field.



Now viewing the updated details of the patient.



2] Compiling and deploying the Doctor's smart contract

Problem Statement:

Developing the Doctor's smart contract which will enable authorized doctors to retrieve patient records, leveraging the patient's address and a unique patient identifier. Subsequently, doctors can update medication details and clinical notes, with these modifications seamlessly reflected within the Patient Smart Contract, ensuring data consistency across both contracts.

Programs for both the Smart Contracts:

Patient Smart Contract:

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;


contract PatientInfo

{

    address public owner;

    address public doctorContractAddress;


    constructor()

    {

        owner = msg.sender;

    }


    struct MedicalRecord

    {

        string name;

        uint256 age;

        string bloodGroup;

        string allergies;

        string medications;

        string surgeries;
```

```
    string physicianNotes;
}

mapping(address => mapping(uint256 => MedicalRecord)) private records;
mapping(address => address[]) private authorizedDoctors;

event MedicalRecordAdded(address indexed patient, uint256 indexed patientId);
event MedicalRecordUpdated(address indexed patient, uint256 indexed patientId);
event DoctorAuthorized(address indexed patient, address indexed doctor);

modifier onlyOwner()
{
    require(msg.sender == owner, "Only owner can call");
    _;
}

function setDoctorContract(address _doctorContract) external onlyOwner
{
    doctorContractAddress = _doctorContract;
}

function addMedicalRecord(
    uint256 patientId,
    string memory name,
    uint256 age,
    string memory bloodGroup,
    string memory allergies,
    string memory medications,
    string memory surgeries,
    string memory physicianNotes
```

```
    ) public
    {
        records[msg.sender][patientId] = MedicalRecord(
            name, age, bloodGroup, allergies, medications, surgeries, physicianNotes
        );
        emit MedicalRecordAdded(msg.sender, patientId);
    }

    // Modified function signature to add doctorCaller
    function updateMedicalRecord(
        address patient,
        uint256 patientId,
        string memory name,
        uint256 age,
        string memory bloodGroup,
        string memory allergies,
        string memory medications,
        string memory surgeries,
        string memory physicianNotes,
        address doctorCaller
    ) public
    {
        // Authorization logic remains the same
        if (msg.sender == patient)
        {
            // patient themselves can update
        } else if (msg.sender == doctorContractAddress)
        {
            require(isDoctorAuthorized(patient, doctorCaller), "Doctor not authorized by
patient");
        } else
```

```
    {
        revert("Not authorized to update this record");
    }

    MedicalRecord storage record = records[patient][patientId];

    if (bytes(name).length > 0) record.name = name;
    if (age > 0) record.age = age;
    if (bytes(bloodGroup).length > 0) record.bloodGroup = bloodGroup;
    if (bytes(allergies).length > 0) record.allergies = allergies;
    if (bytes(medications).length > 0) record.medications = medications;
    if (bytes(surgeries).length > 0) record.surgeries = surgeries;
    if (bytes(physicianNotes).length > 0) record.physicianNotes = physicianNotes;

    emit MedicalRecordUpdated(patient, patientId);
}

function authorizeDoctor(address doctor) public
{
    authorizedDoctors[msg.sender].push(doctor);
    emit DoctorAuthorized(msg.sender, doctor);
}

function isDoctorAuthorized(address patient, address doctor) public view returns (bool)
{
    address[] memory doctors = authorizedDoctors[patient];
    for (uint i = 0; i < doctors.length; i++)
    {
        if (doctors[i] == doctor)
        {
```

```
            return true;
        }
    }
    return false;
}

function getRecord(address patient, uint256 patientId)
    external
    view
    returns (MedicalRecord memory)
{
    return records[patient][patientId];
}

// New function: returns record fields line-wise as tuple
function getRecordLineWise(address patient, uint256 patientId)
    external
    view
    returns (
        string memory name,
        uint256 age,
        string memory bloodGroup,
        string memory allergies,
        string memory medications,
        string memory surgeries,
        string memory physicianNotes
    )
{
    MedicalRecord storage record = records[patient][patientId];
    return (
```

```
            record.name,
            record.age,
            record.bloodGroup,
            record.allergies,
            record.medications,
            record.surgeries,
            record.physicianNotes
        );
    }
}
```

Doctor Smart Contract:

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

interface IPatientInfo
{
    struct MedicalRecord
    {
        string name;
        uint256 age;
        string bloodGroup;
        string allergies;
        string medications;
        string surgeries;
        string physicianNotes;
    }
    function isDoctorAuthorized(address patient, address doctor) external view returns (bool);

    function getRecord(address patient, uint256 patientId)
        external
```

```
        view
        returns (MedicalRecord memory);

    // Modified function signature with doctorCaller
    function updateMedicalRecord(
        address patient,
        uint256 patientId,
        string memory name,
        uint256 age,
        string memory bloodGroup,
        string memory allergies,
        string memory medications,
        string memory surgeries,
        string memory physicianNotes,
        address doctorCaller
    ) external;
}
contract Doctor
{
    IPatientInfo public patientContract;

    constructor(address _patientContract)
    {
        patientContract = IPatientInfo(_patientContract);
    }

    modifier onlyAuthorized(address patient)
    {
        require(
            patientContract.isDoctorAuthorized(patient, msg.sender),
```

```
        "Doctor not authorized by patient"
    );
    _;
}
event RecordUpdated(address indexed patient, uint256 indexed patientId);
function viewPatientRecord(address patient, uint256 patientId)
    public
    view
    onlyAuthorized(patient)
    returns (
        string memory name,
        uint256 age,
        string memory bloodGroup,
        string memory allergies,
        string memory medications,
        string memory surgeries,
        string memory physicianNotes
    )
{
    IPatientInfo.MedicalRecord memory record =
        patientContract.getRecord(patient, patientId);
    return (
        record.name,
        record.age,
        record.bloodGroup,
        record.allergies,
        record.medications,
        record.surgeries,
        record.physicianNotes
    );
```

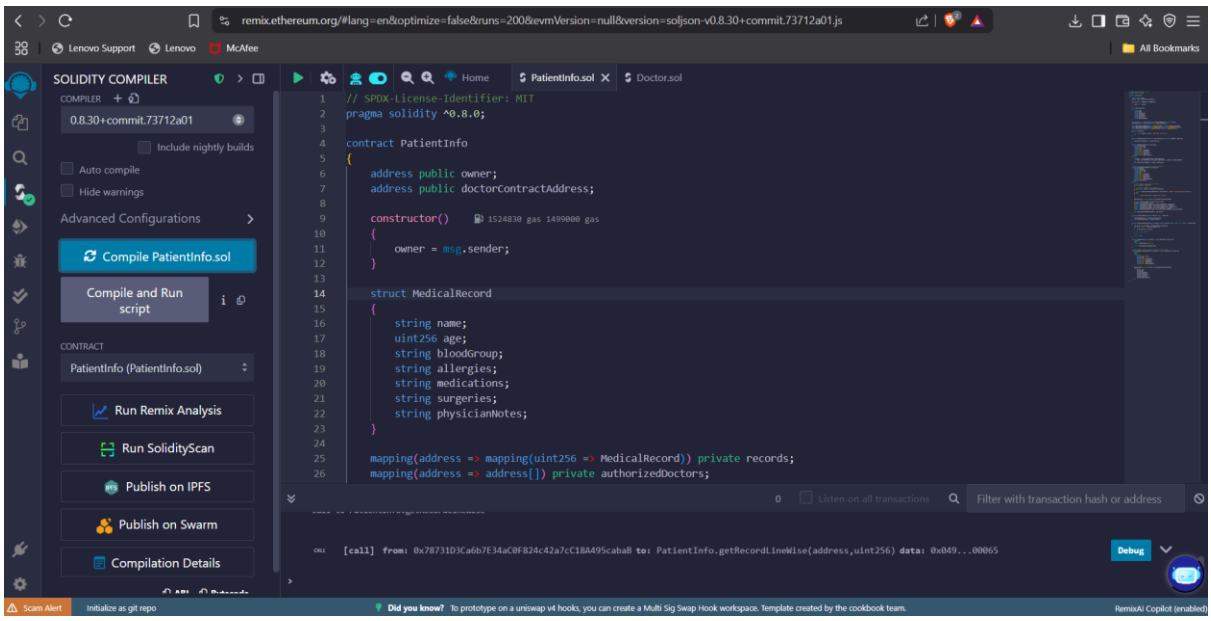
```
}
function updateMedicationsAndNotes(
    address patient,
    uint256 patientId,
    string memory newMedications,
    string memory newPhysicianNotes
) public onlyAuthorized(patient)
{
    IPatientInfo.MedicalRecord memory record =
        patientContract.getRecord(patient, patientId);

    patientContract.updateMedicalRecord(
        patient,
        patientId,
        record.name,
        record.age,
        record.bloodGroup,
        record.allergies,
        newMedications,
        record.surgeries,
        newPhysicianNotes,
        msg.sender // <-- Pass the doctor's address here
    );

    emit RecordUpdated(patient, patientId);
}
function getMyAddress() public view returns (address)
{
    return msg.sender;
}
```

}

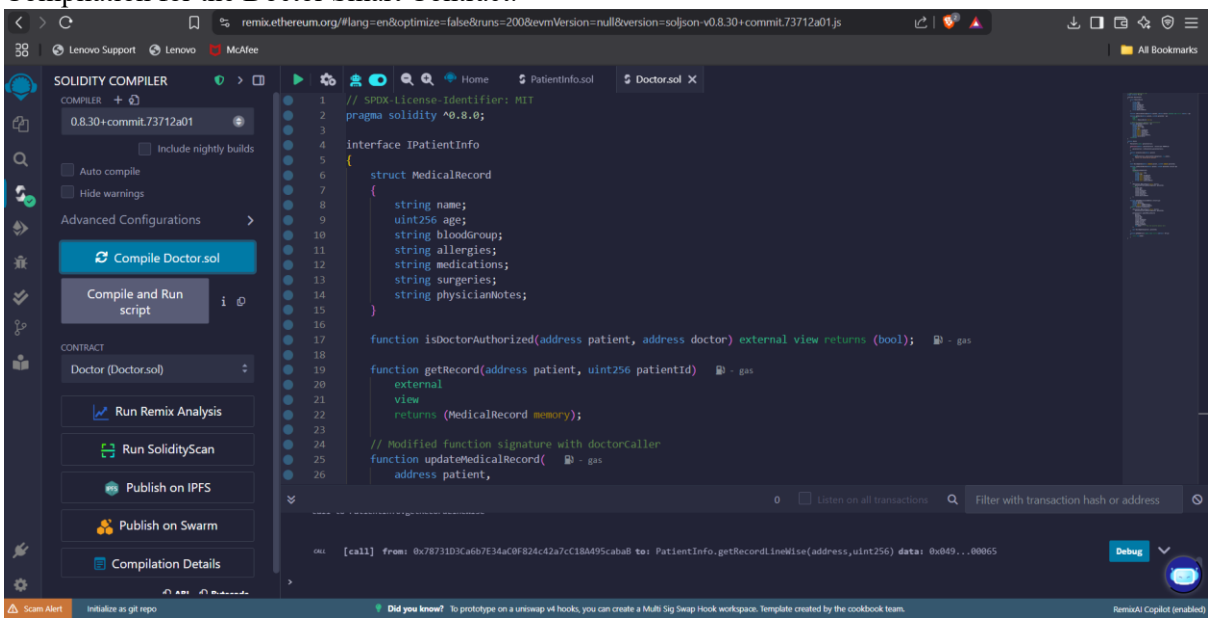
Execution and deployment of the smart contracts:

Step 1: Compiling both the Smart contracts

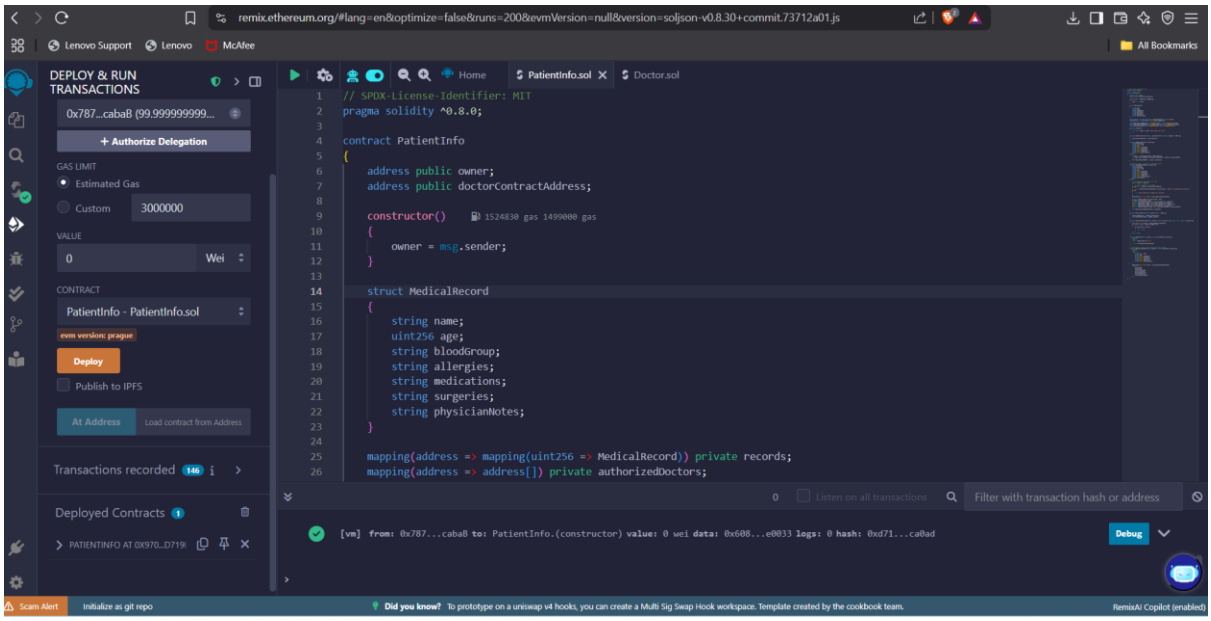
Compilation for the Patient Smart Contract:



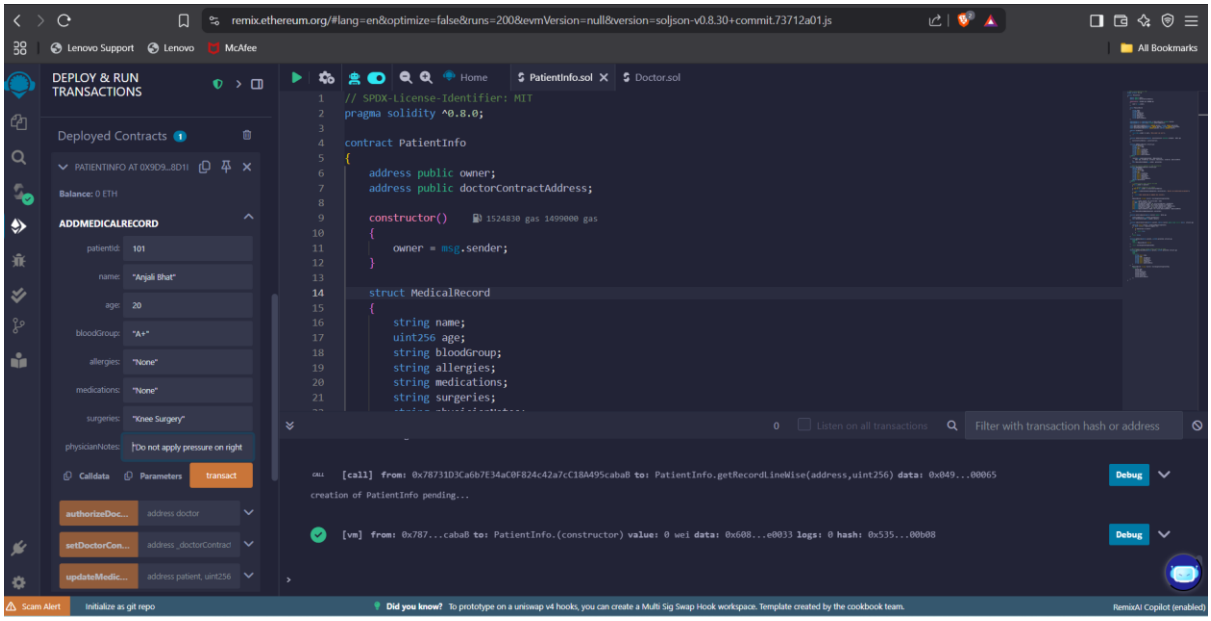
Compilation for the Doctor Smart Contract:



Step 2: Using an account, deploying the patient's contract

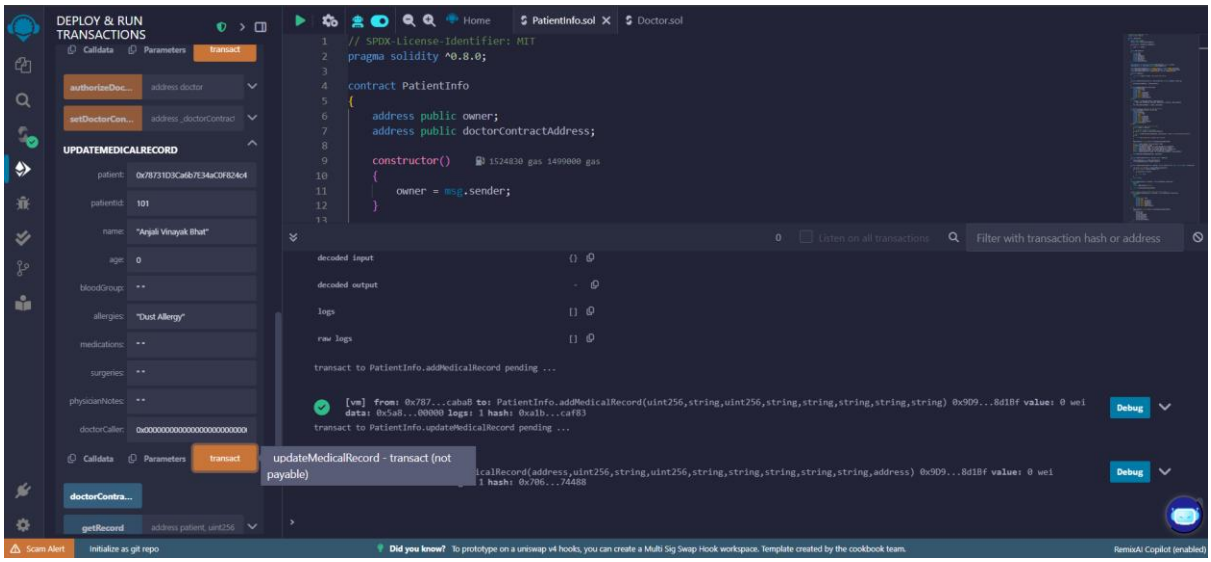


Step 3: Entering the patient's details after the patient's smart contract has been deployed and then transact it.

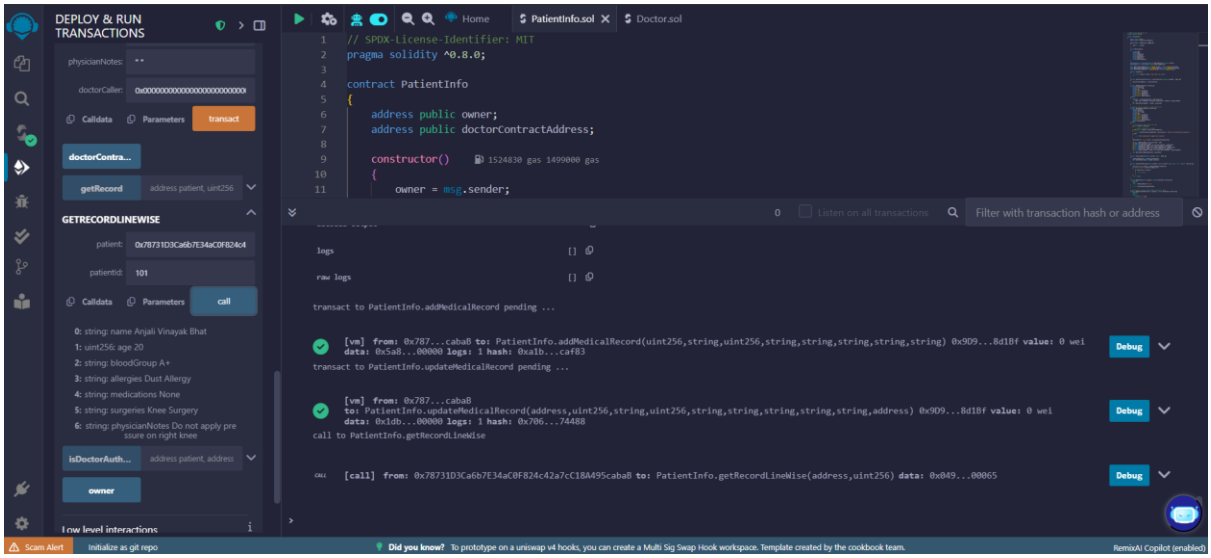


Step 4: Then update the details of the patient. *[note this step is only required if you want to update the details of the patient else you can skip this step]*

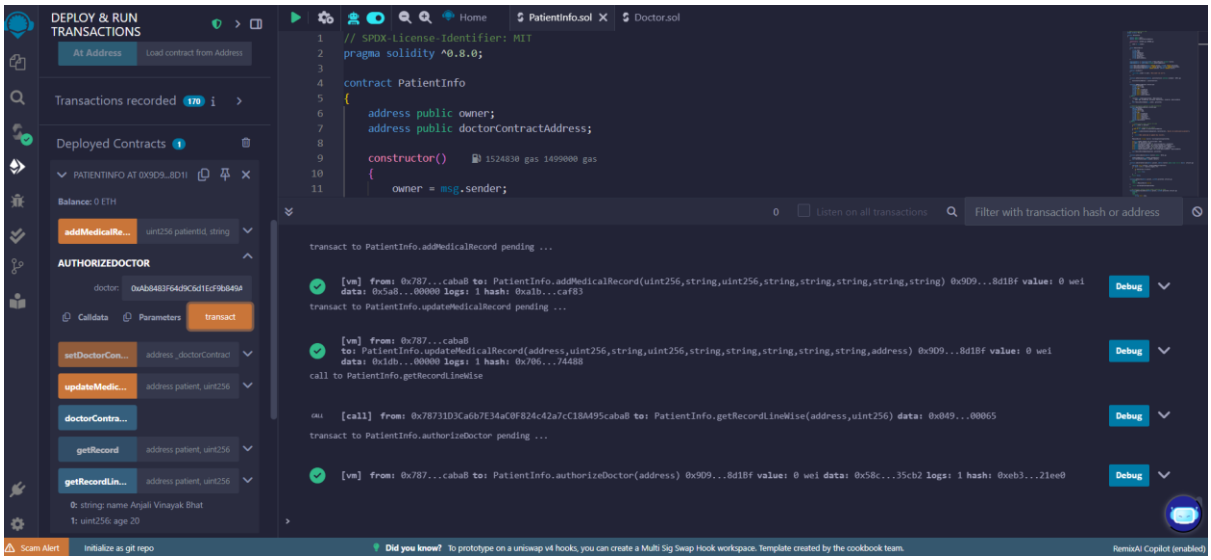
Note: Here only update the fields you want to and keep the rest as none to retrieve from the previous information, even for the doctor's address we set it to null as the patient is calling from their address to make the change.



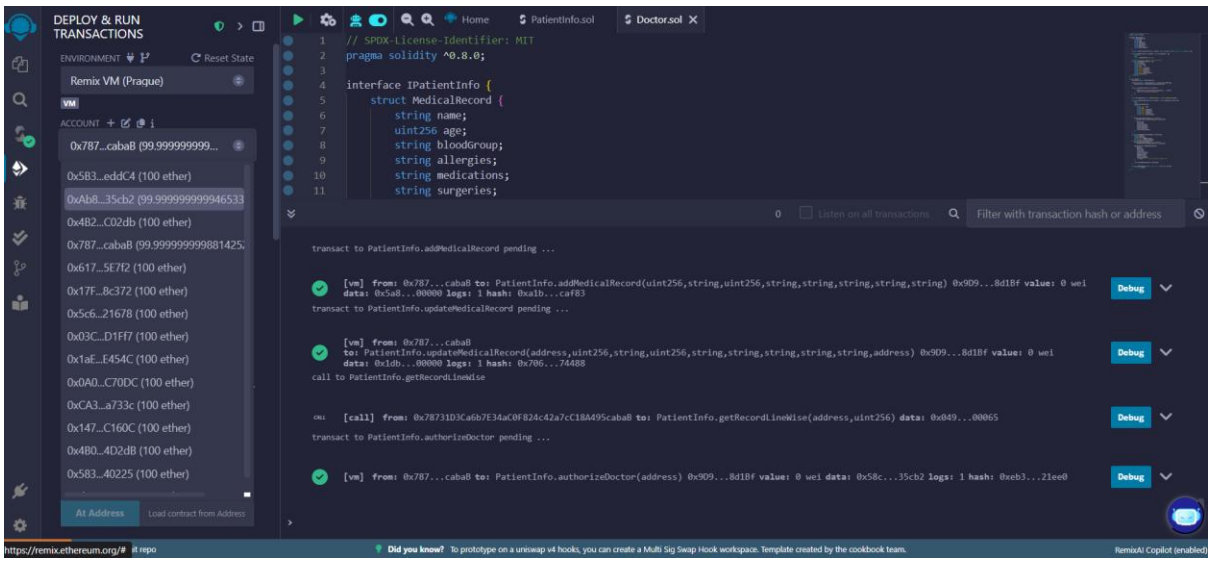
Step 5: Now we call the get record line by line function to see if the update is made.



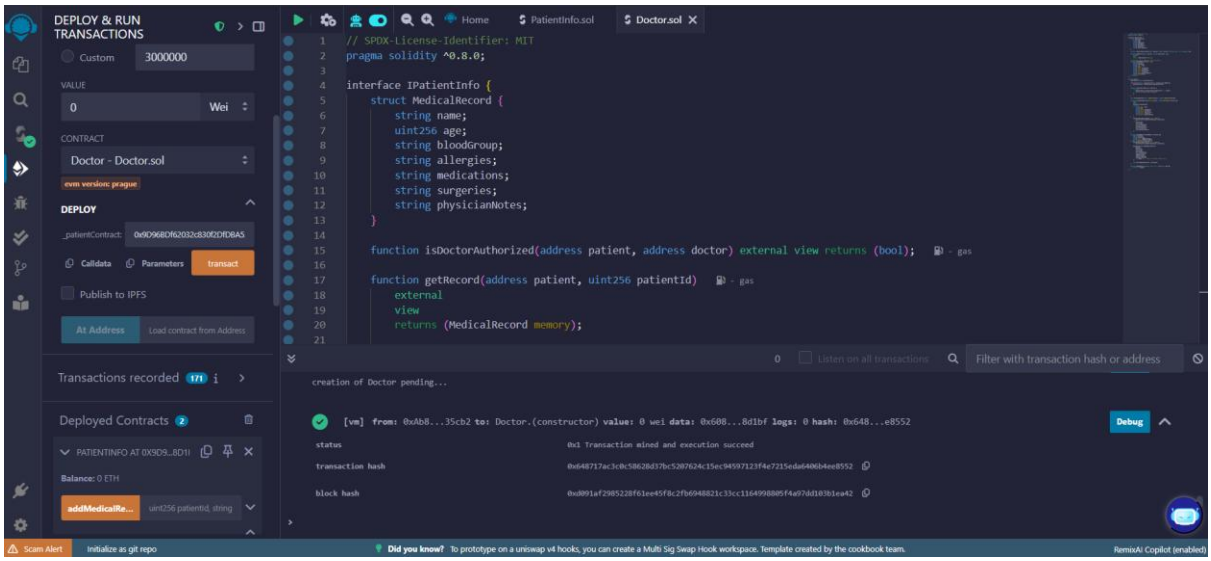
Step 6: Now use another account address for deploying the doctor's contract



Step 7: Now taking the Doctor's contract address, deploy the doctor's smart contract to generate doctor's contract address.

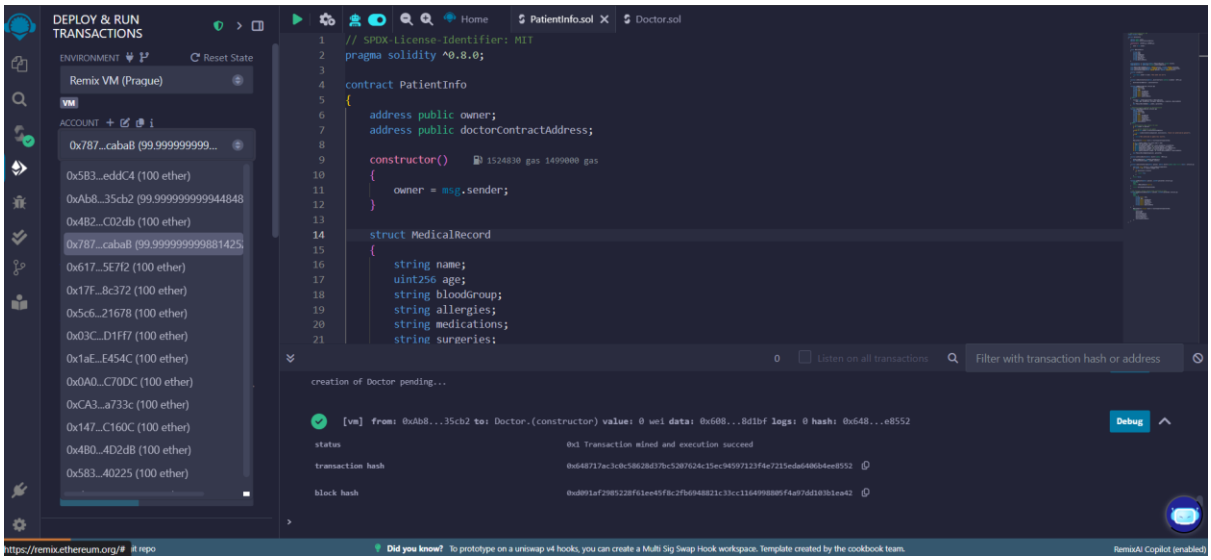


Now we deploy the doctor's smart contract by entering the Patient's contract address

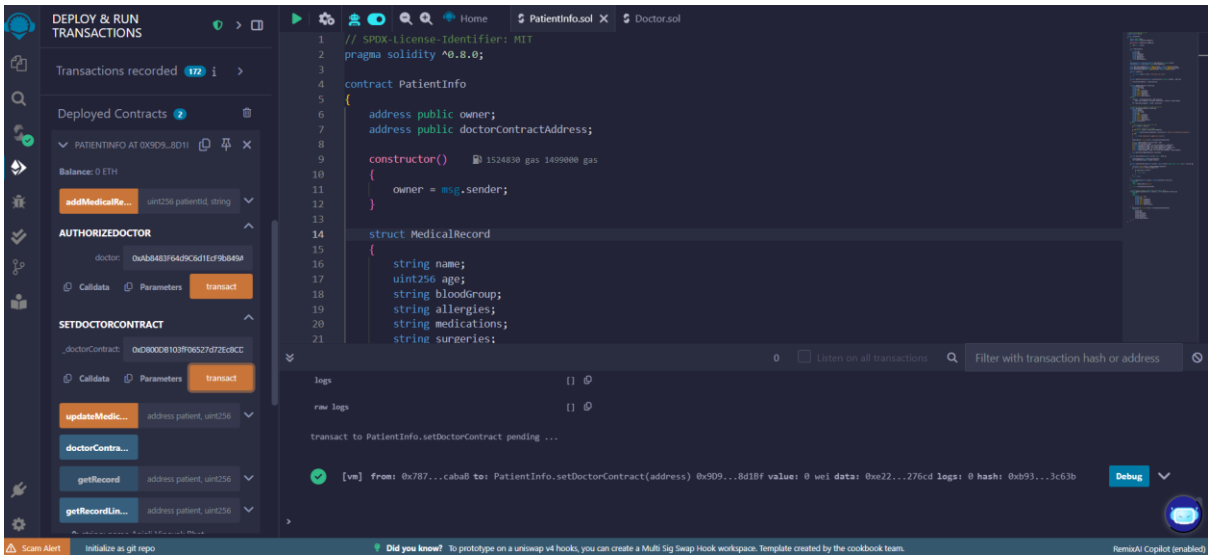


Step 8: Now move back to the patient's smart contract to authorise the doctor's contract address.

Here use the patient's account address

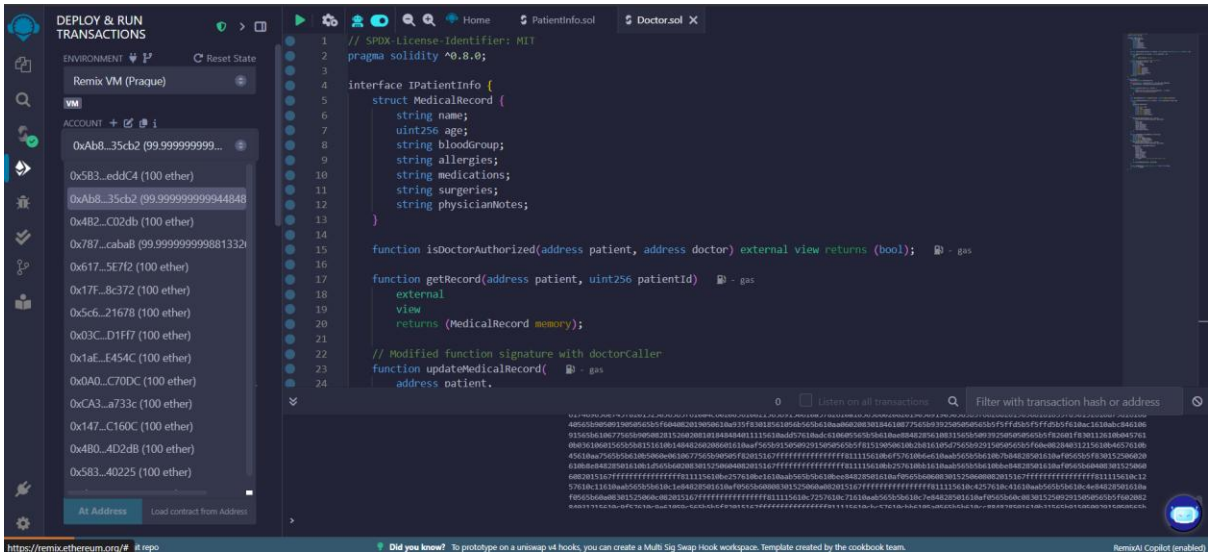


Now authorize the doctor's contract address

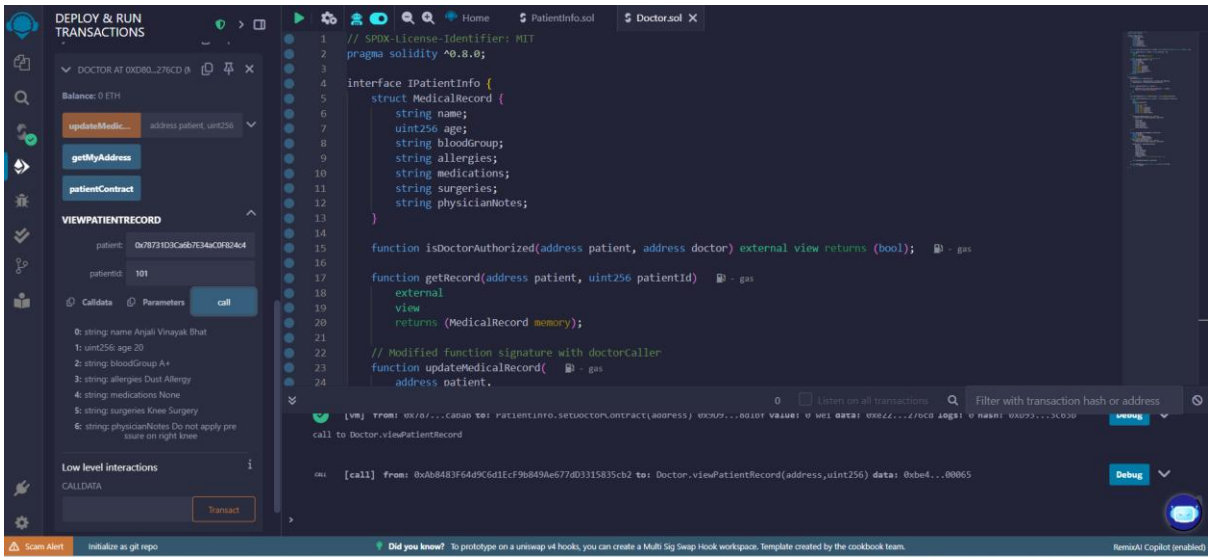


Step 9: Update the patient's details (medications and physician notes) via the doctor's smart contract.

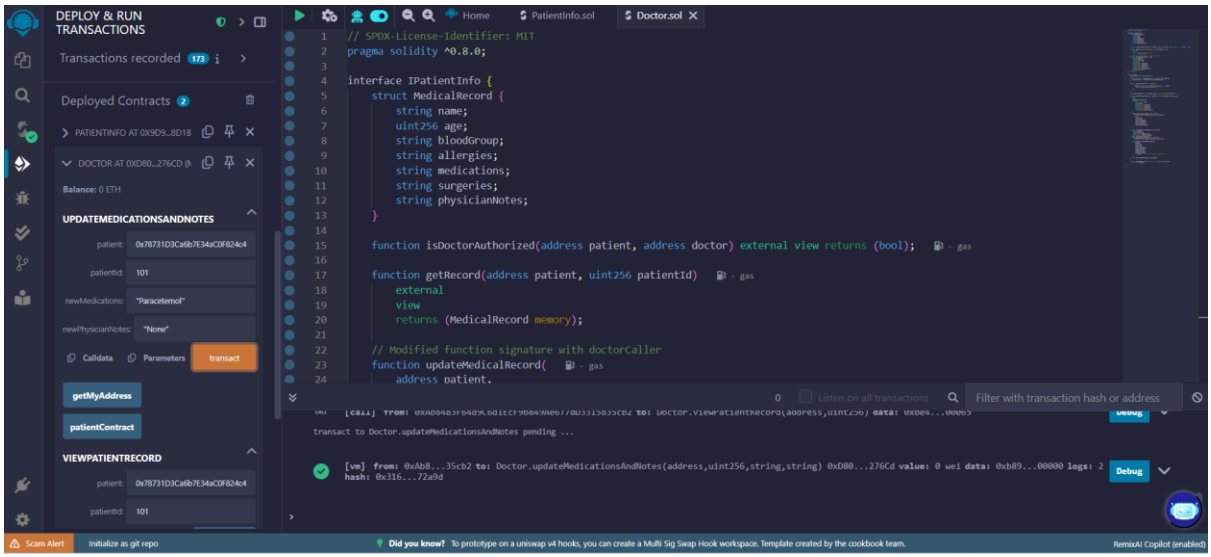
Now switch to the Doctor's smart contract



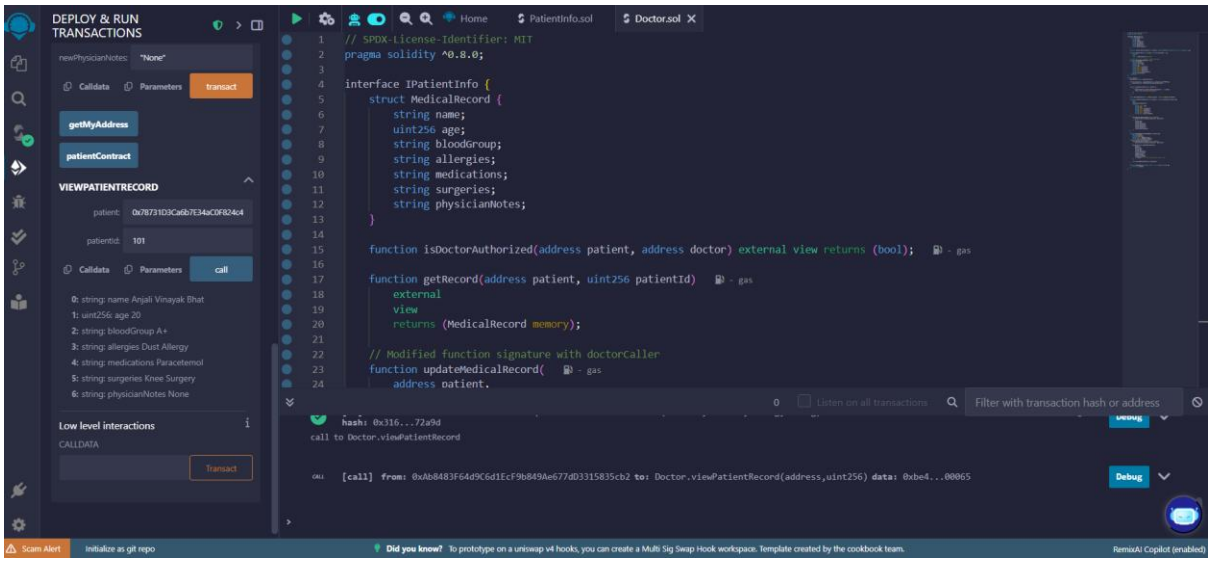
Here first view the records of the patient using their address and patientID



Now update the patient's details (medications and physician notes)

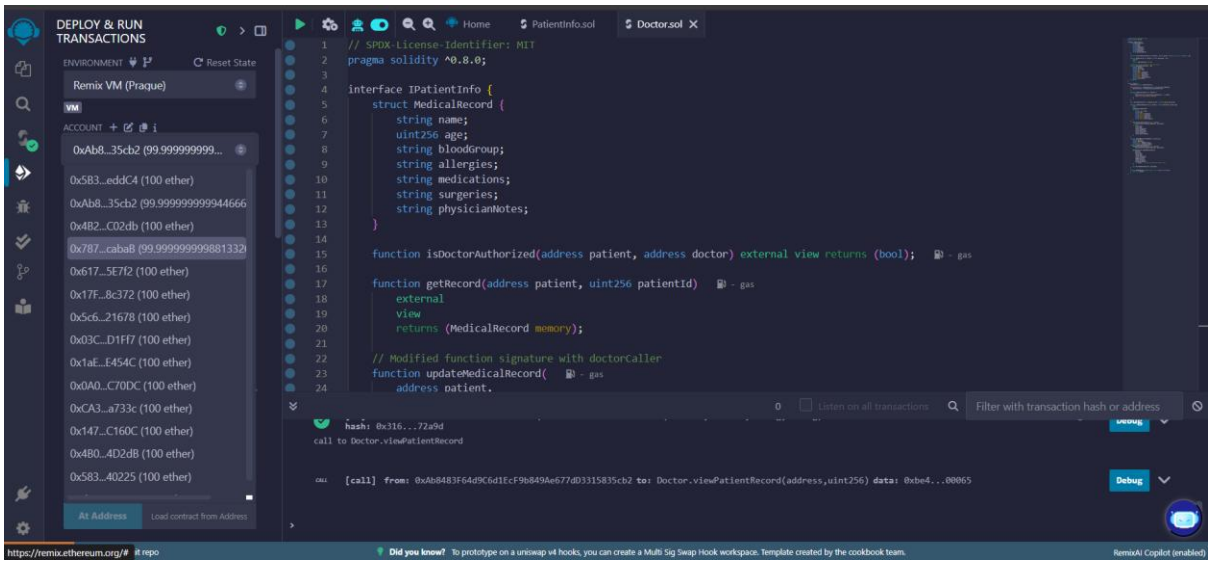


Now view the changes made to the pateint's medical record in the doctor's smart contract itself

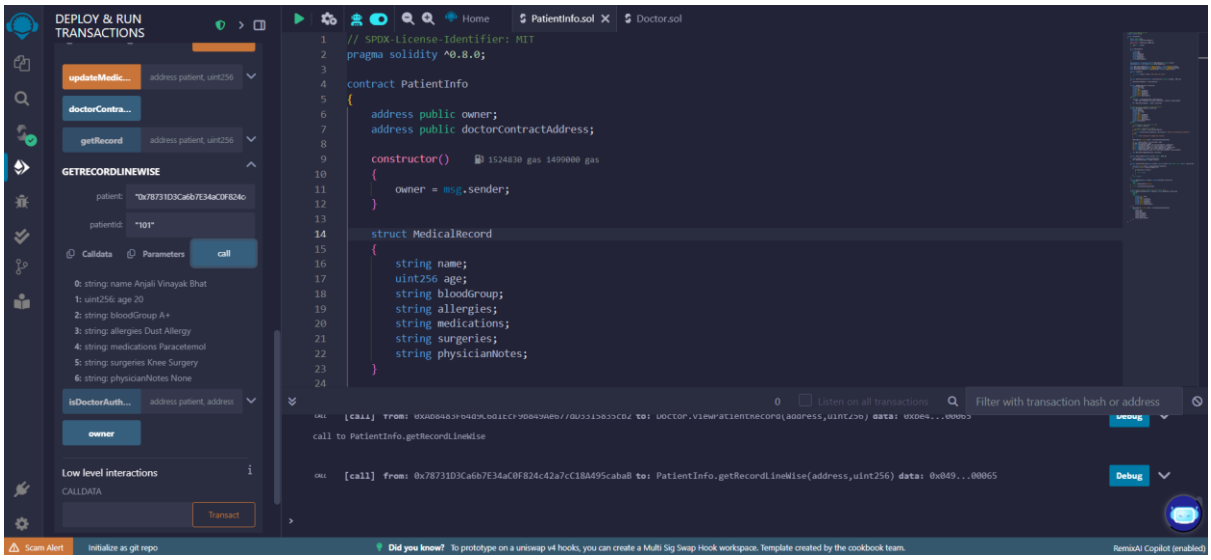


Step 10: Now view the changes made into the patient's medical records in the patient's smart contract.

Switch the address to the patient's address and move to the patient's smart contract now.



The changes are reflected in the patient's smart contract as well



There by this we can see the process of a smart contract calling another smart contract.

Now move to the second phase of Vulnerabilities testing, deploying vulnerabilites and mitigating them.

Analysis of the code

Strengths

Role Separation

1. *Admin (owner)* has control over setting the doctor contract.
2. Patients can:
    - Add their own records.
    - Authorize doctors.
3. Doctors (via a separate contract) can only update authorized patient records.

Access Control

1. onlyOwner modifier protects critical functions like setDoctorContract.
2. isDoctorAuthorized ensures doctors can't access patient data without permission.
3. msg.sender checks prevent unauthorized record manipulation.

Extensibility & Decoupling

1. Use of the IPatientInfo interface allows other contracts (like Doctor) to interact modularly with PatientInfo.

Events

1. Useful events (MedicalRecordAdded, DoctorAuthorized, etc.) are emitted for key actions — this aids off-chain monitoring and auditing.

Walk-through: Day-to-day interactions in a busy clinic

Registration burst

1. 1 000 patients call authorizeDoctor(hospitalProxy) once.
2. Gas ≈ 25 k per call → 25 M gas total. Fine.
3. Arrays hold 1 doctor each—look-ups are cheap.

Growth to 30 doctors per patient

1. Each new specialist is pushed. Update visits now cost extra gas: isDoctorAuthorized loops 30 entries (~1 300 gas). Still manageable, but trending up.

If Hospital adds second proxy (surgery dept.)

1. Each patient now keeps two addresses.

2. You deploy a new DoctorSurgery contract; patients must call authorizeDoctor().
3. If 10 000 patients forget, their surgeon's update fails = help-desk nightmare.

Contention scenario

1. Cardiologist (tx1) updates meds at block T.
2. Immediately, GP (tx2) updates allergies, reading stale meds.
3. tx2 overwrites meds inadvertently. Lost update.

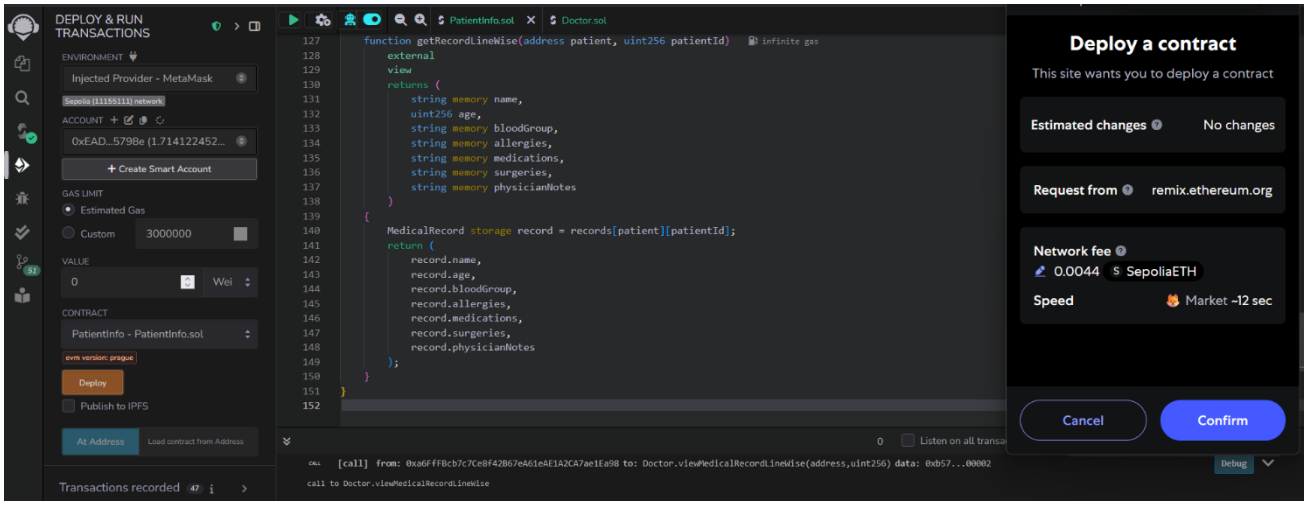
Revocation drama

1. Patient sees abuse, wants to revoke Dr. X urgently.
2. Today: impossible. You need a revokeDoctor method that swaps array element with last, pop().
3. Even with that, Dr. X could mine a tx in the *same* block before revocation is executed.

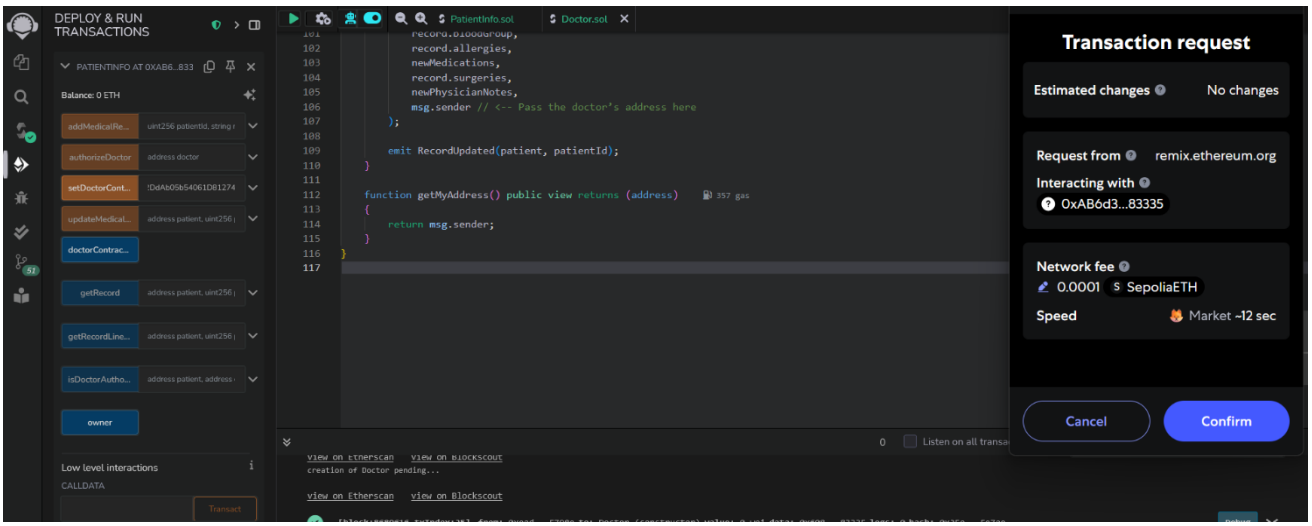
Multiple test cases

1] Basic case with Metamask integrated

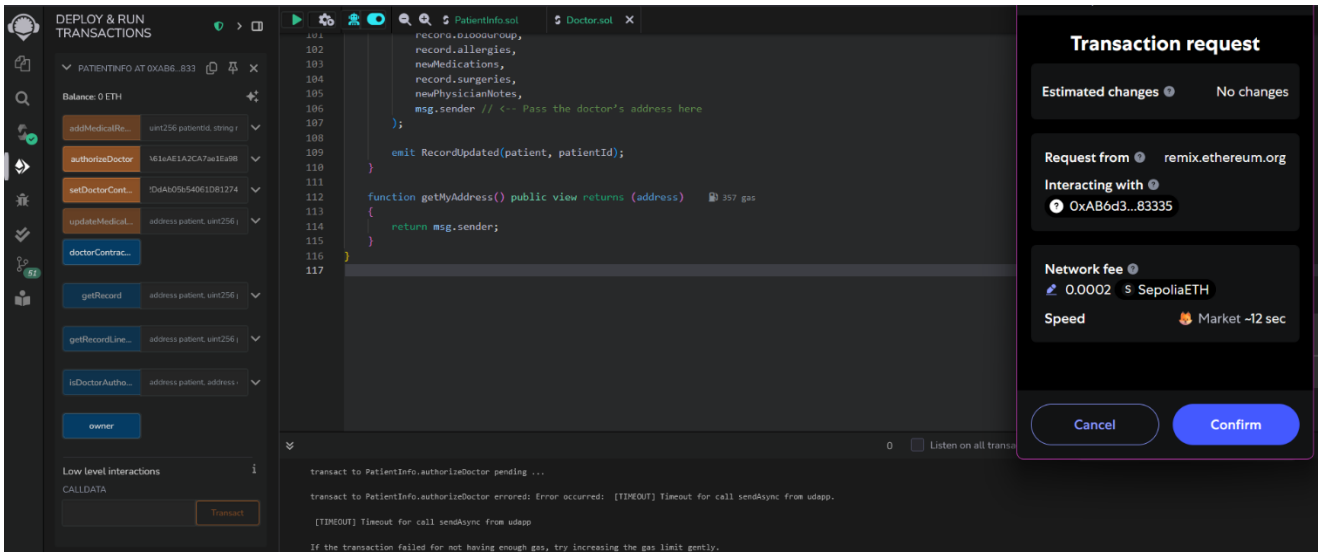
1] Deploying the patient's contract



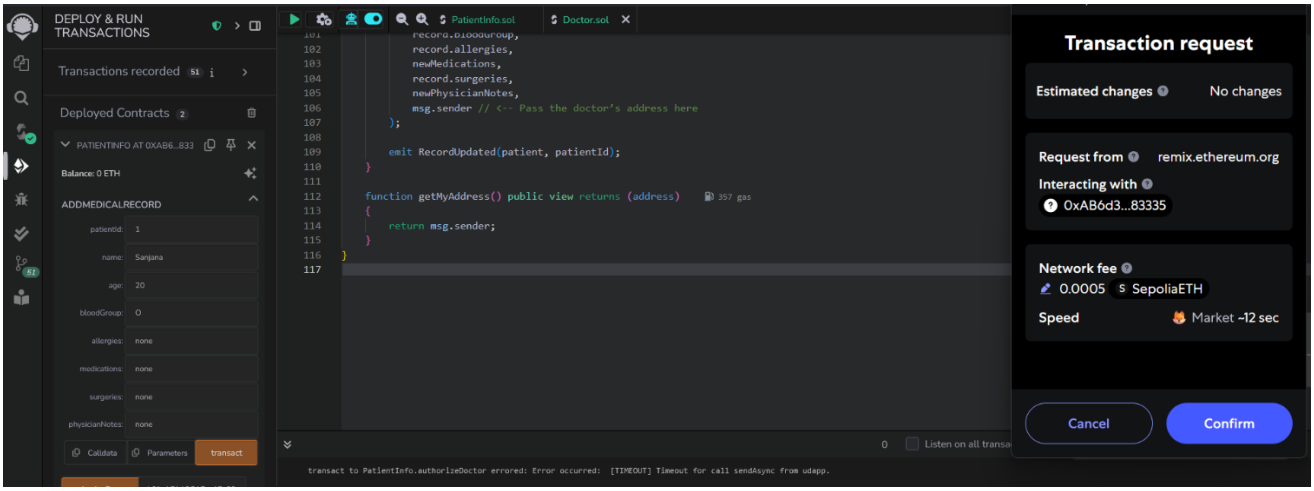
2] Setting doctor's contract in patient's contract



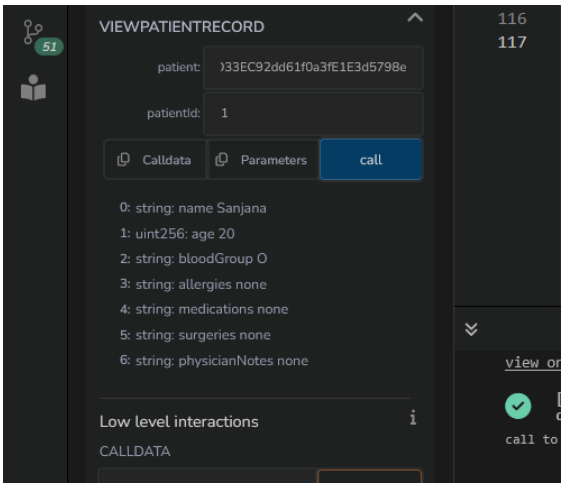
3] Authorizing Doctor’s contract in the patient’s contract



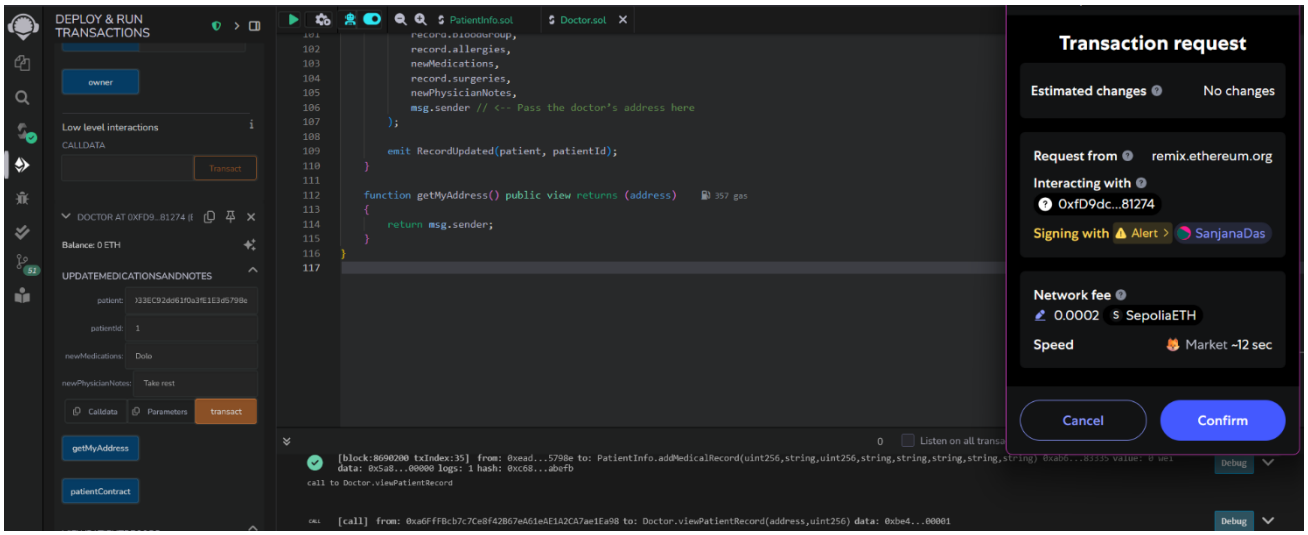
4] Adding patient details in the patient’s contract



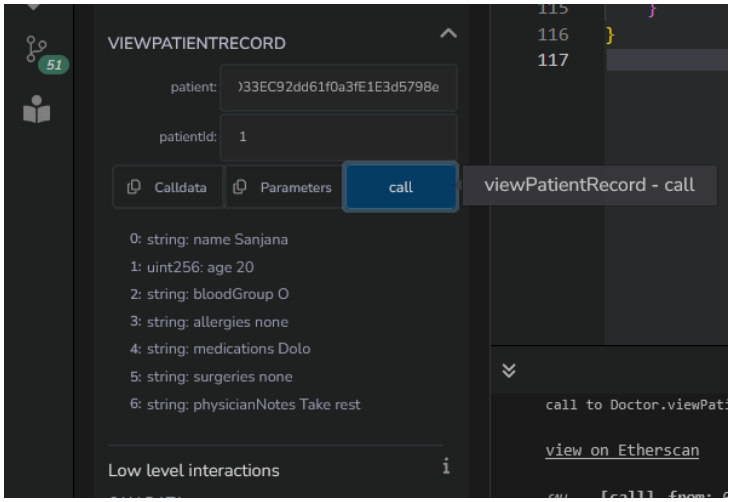
5] Viewing patient details in the doctor’s contract



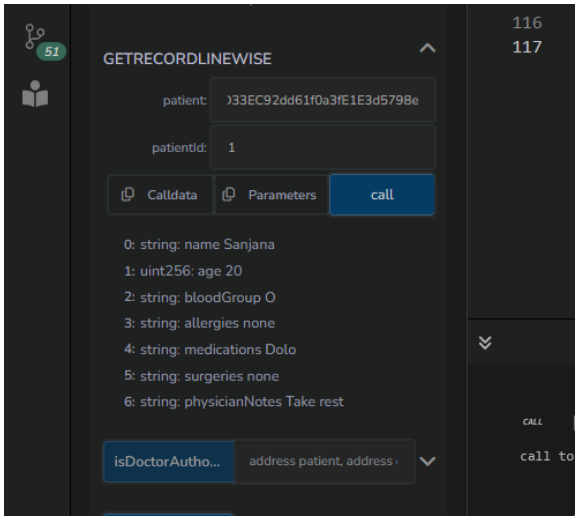
6] Updating patient details on the doctor’s contract



7] Viewing updated patient details on the doctor’s contract

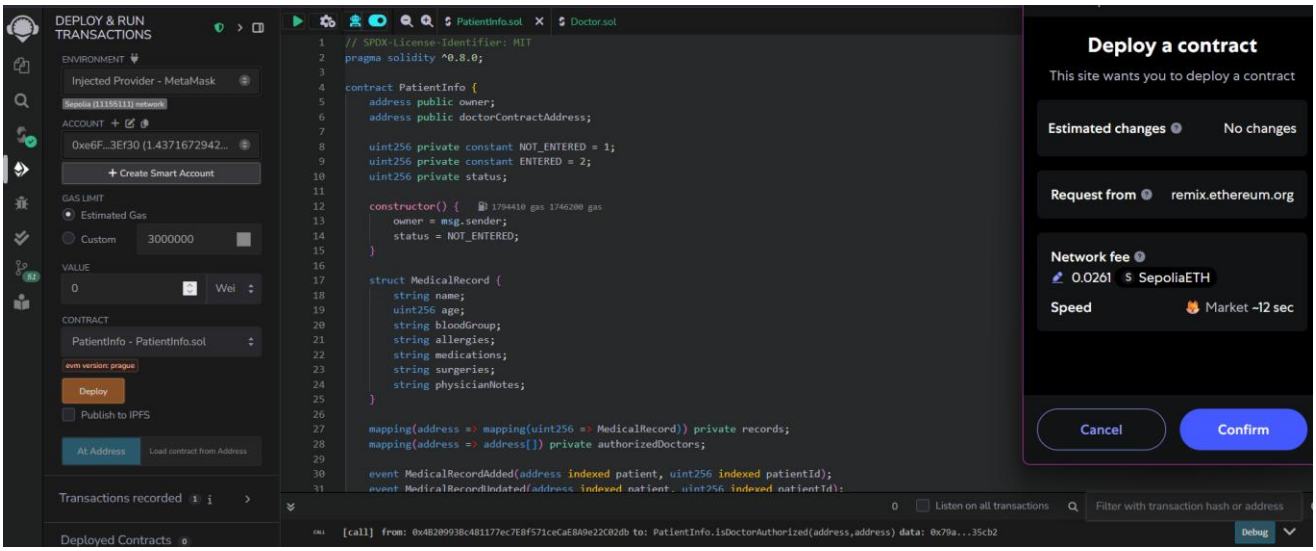


8] Viewing updated patient details on the patient’s contract

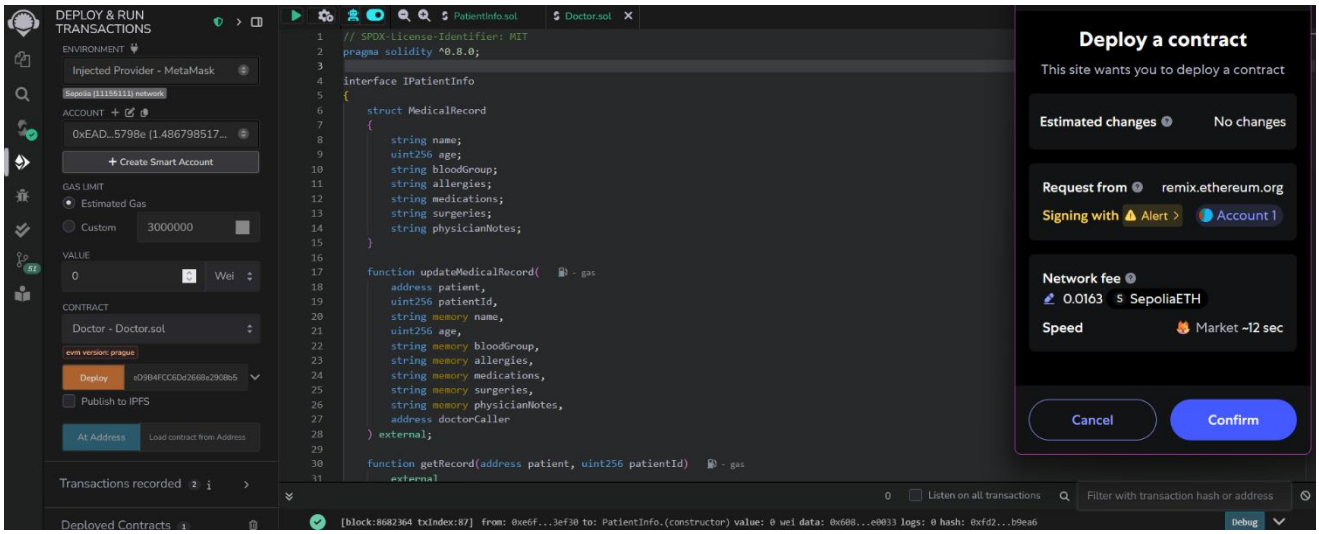


2] Implementation of two patients one doctor

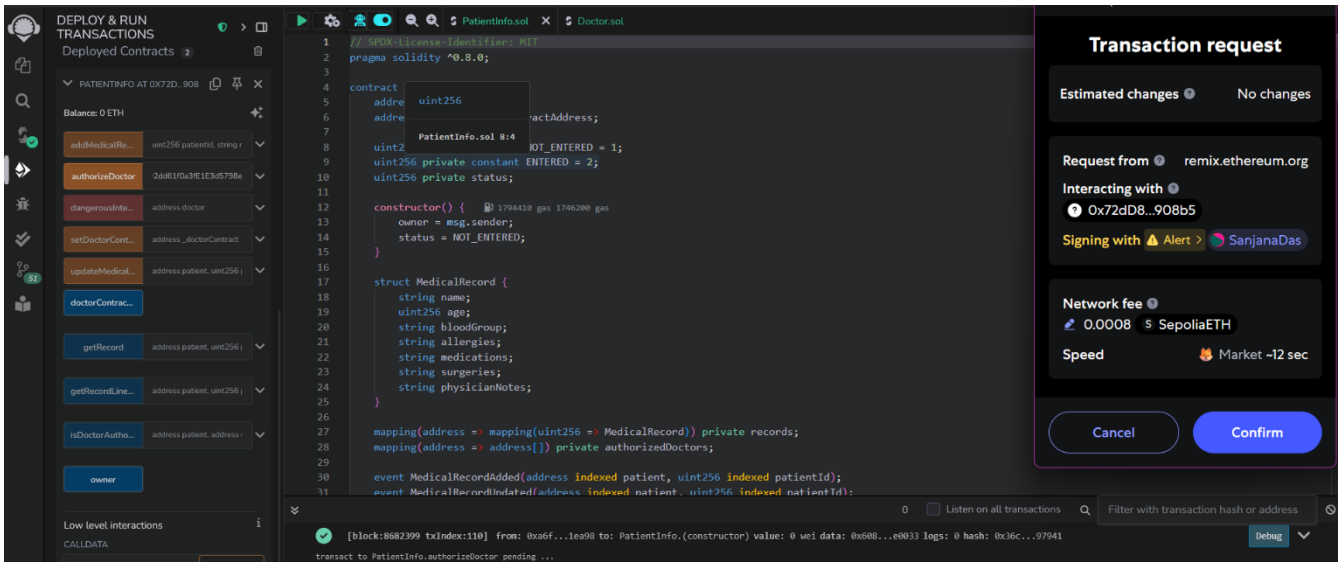
1] Deploying first patient's contract



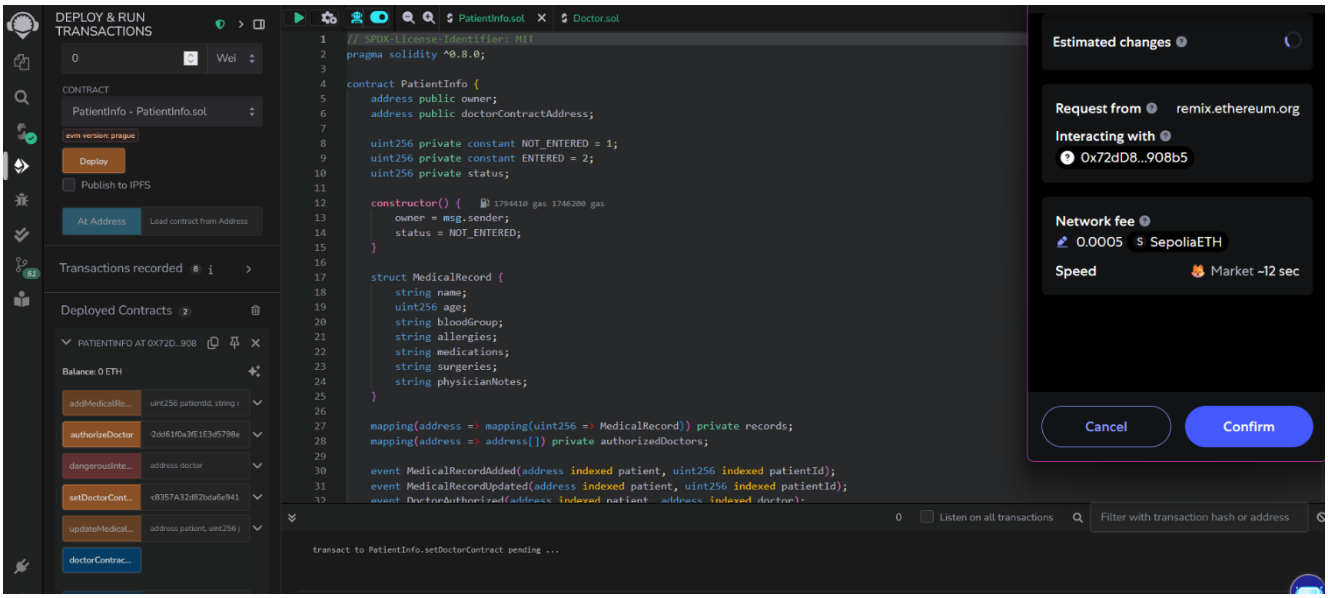
2] Deploying Doctor's contract with patient's address



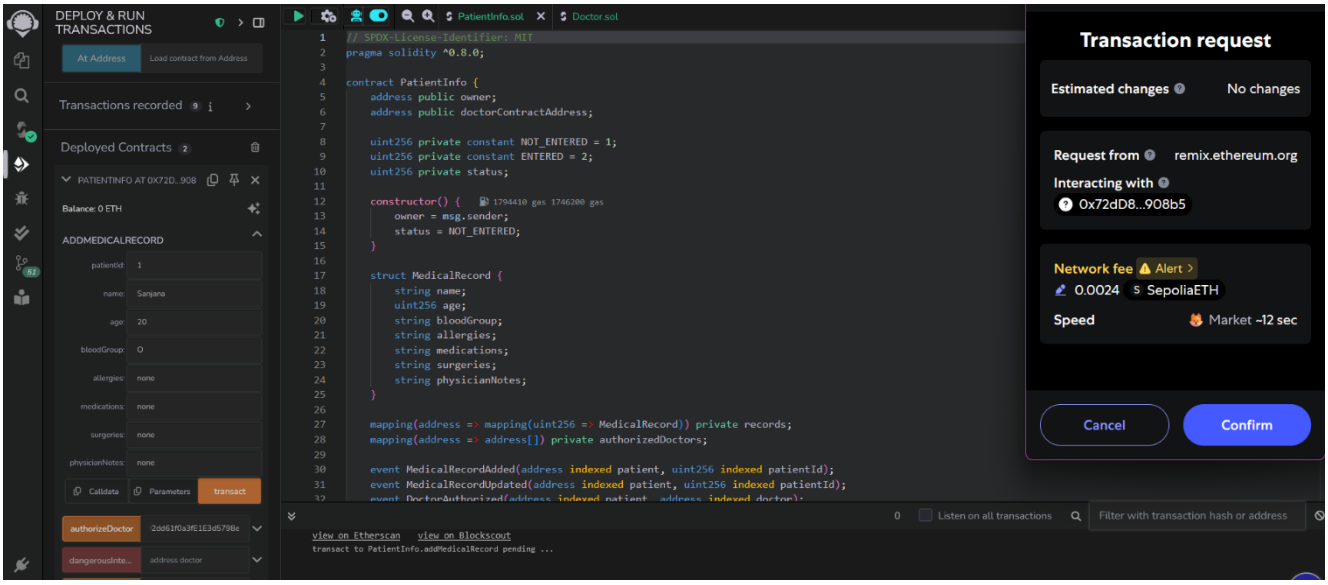
3] Authorizing Doctor to the patient contract



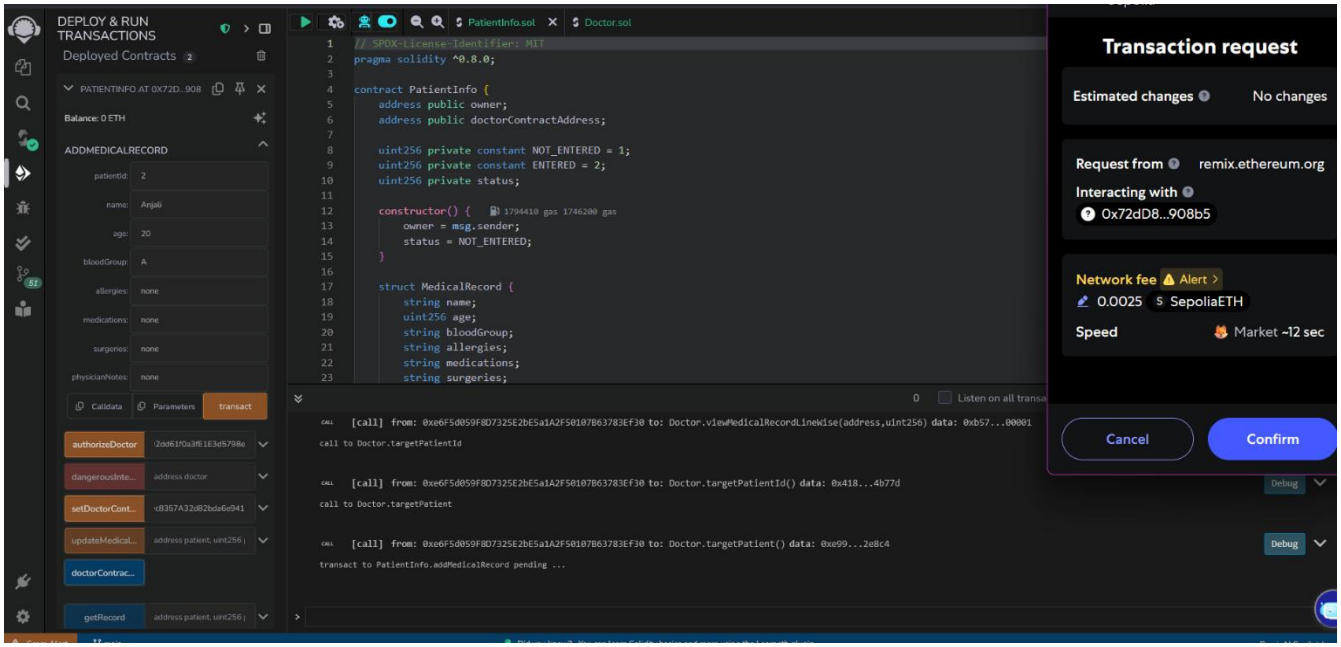
4] Setting the doctor’s contract within the patient’s contract



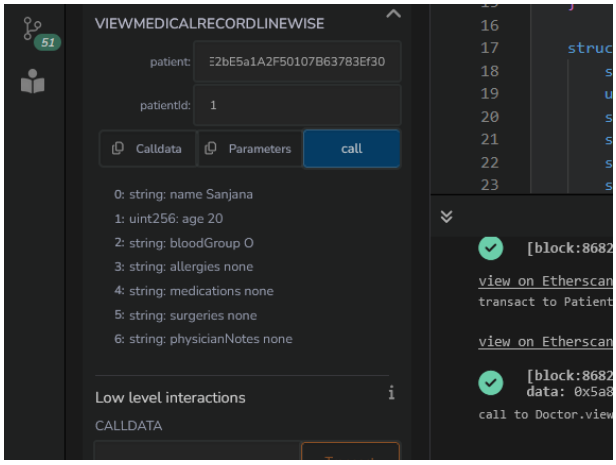
5] Entering the first patient’s details



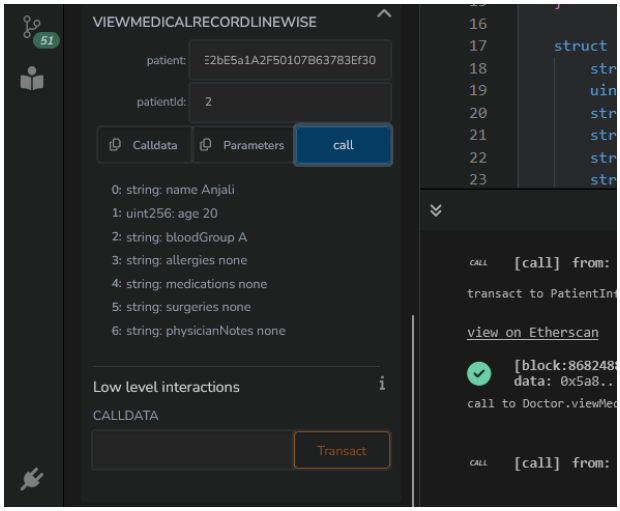
6] Entering the second patient’s details



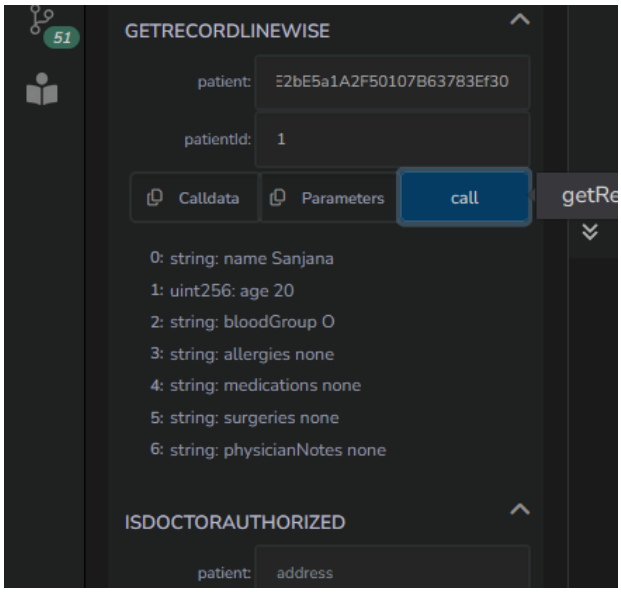
7] Viewing the first patient's details in the doctor's contract



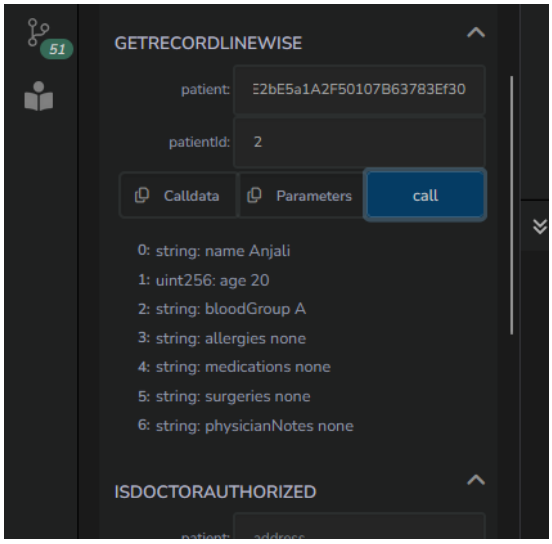
8] Viewing the second patient's details in the doctor's contract



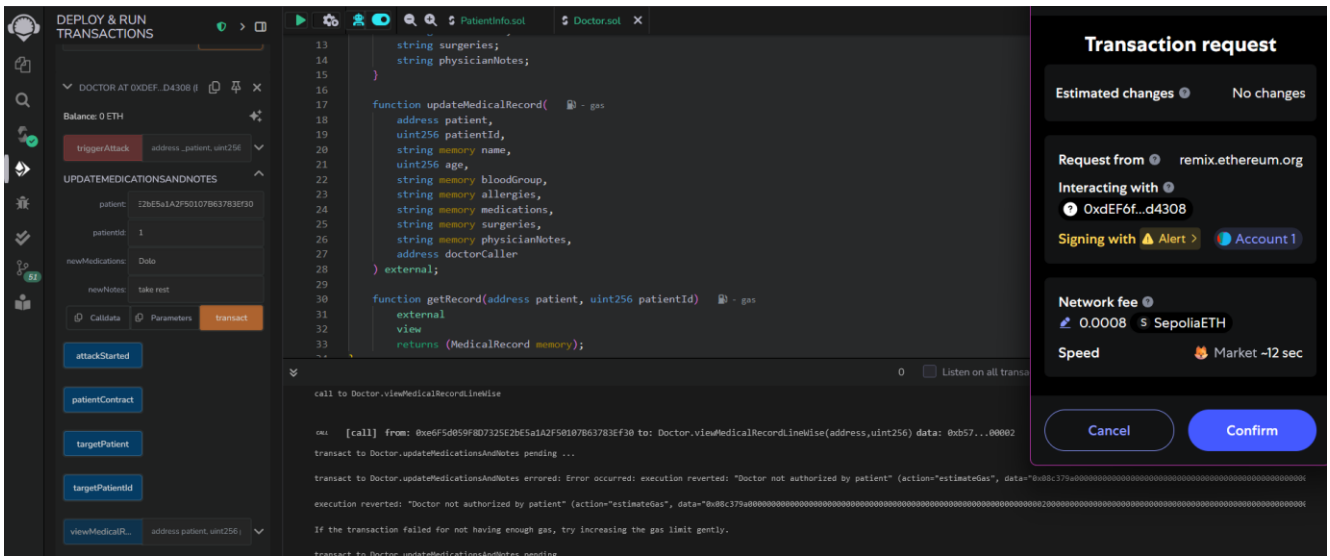
9] Viewing the first patient's details in the patient's contract



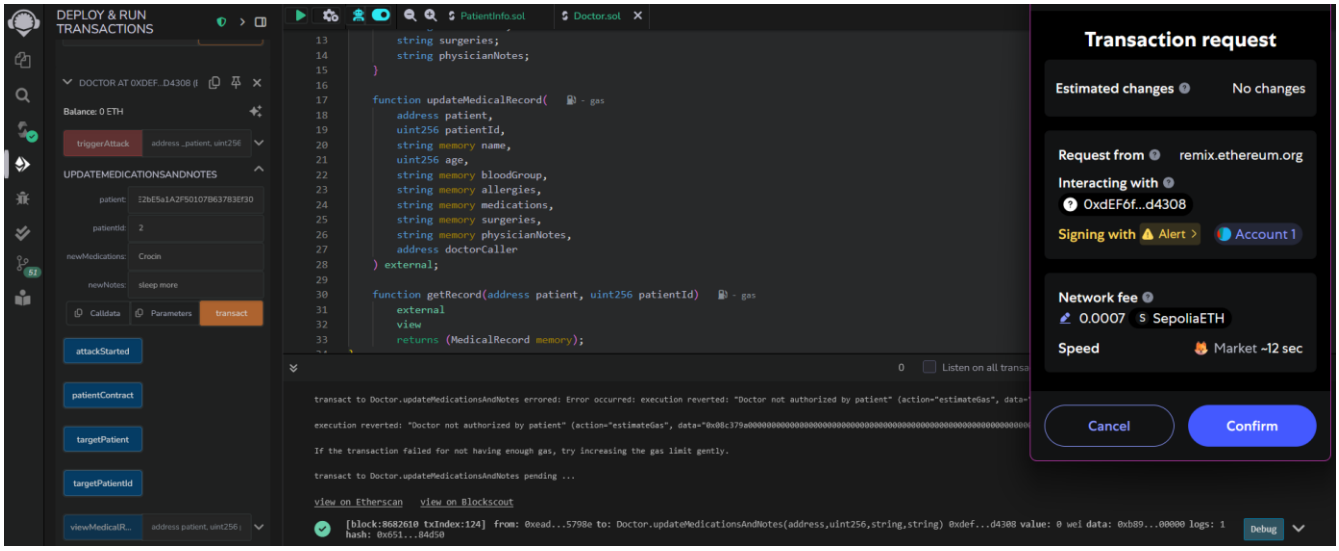
10] Viewing the second patient's details in the patient's contract



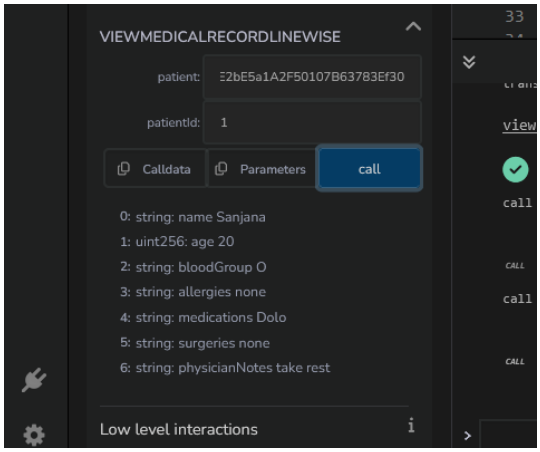
11] Updating the first patients details and viewing it in the doctor's contract



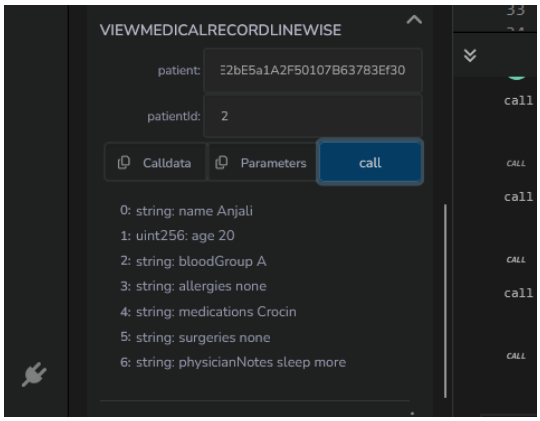
12] Updating the second patients details and viewing it in the doctor's contract



13] Viewing updated medication of the first patient in the doctor's contract

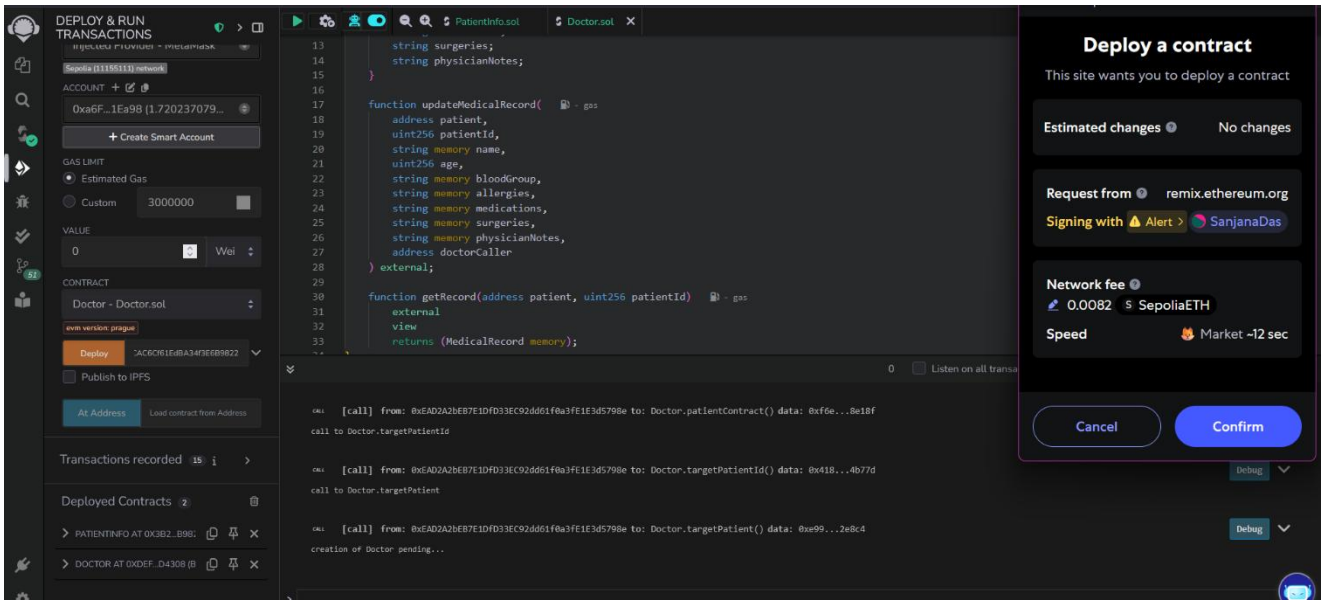


14] Viewing updated medication of the first patient in the doctor's contract

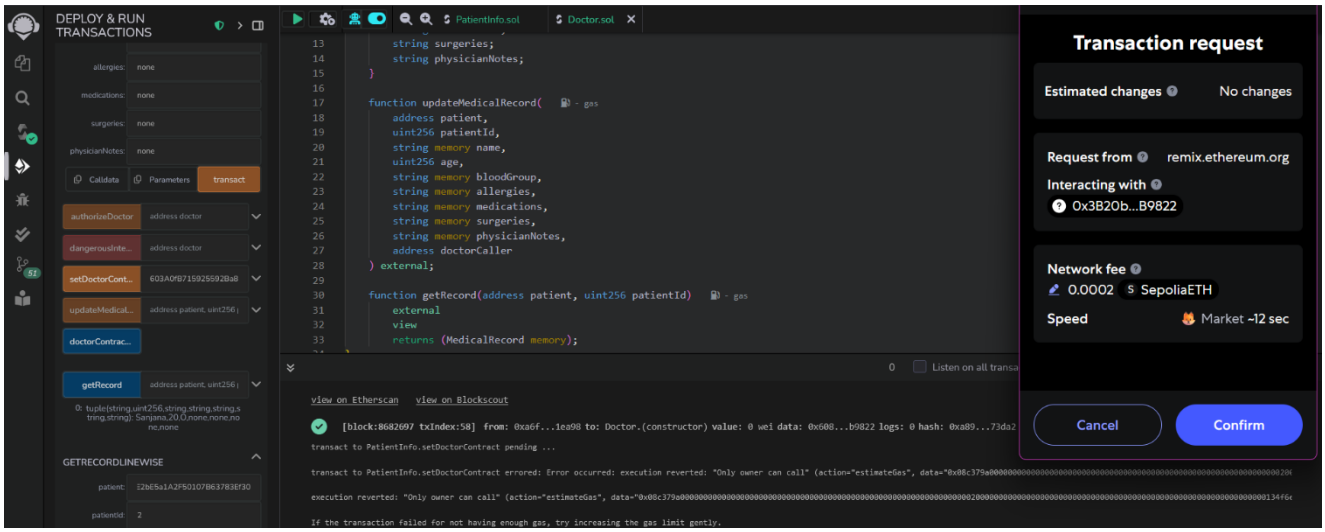


3] Implementation of two patients two doctor

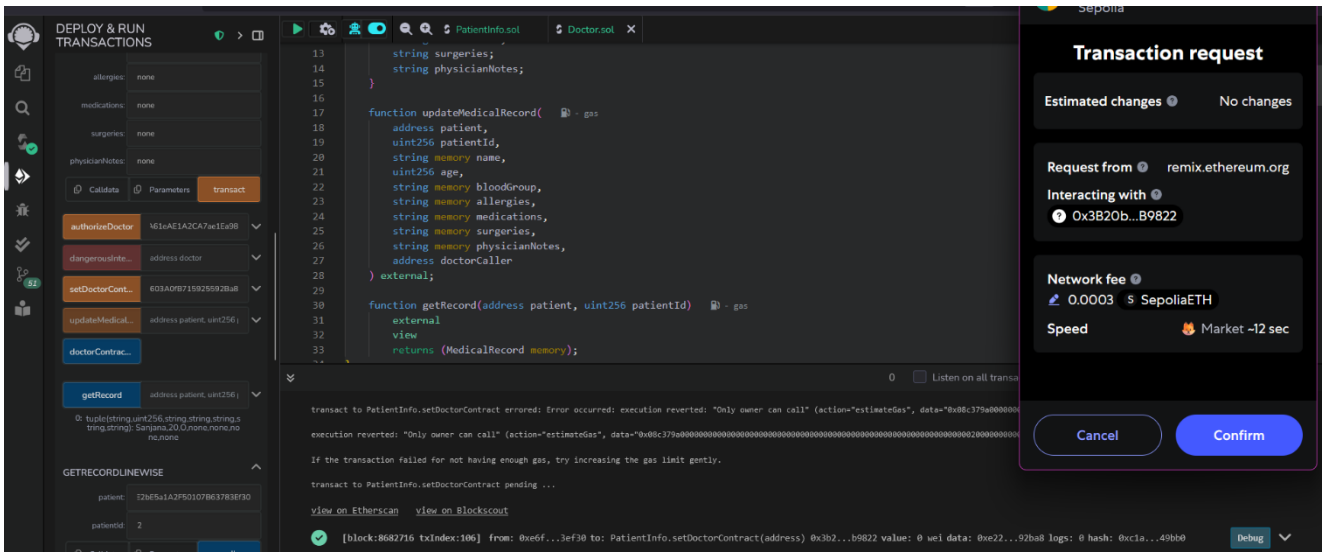
1] Deploying second doctor’s contract



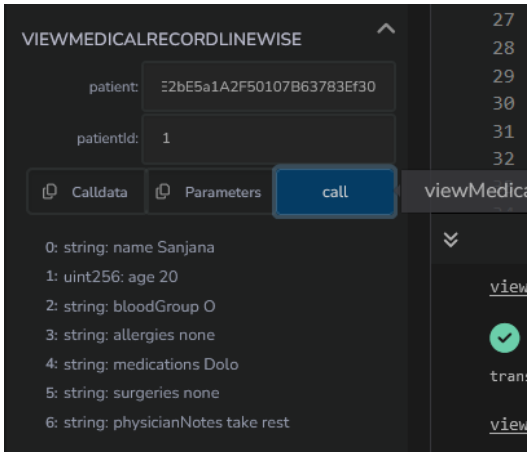
2] Setting the second doctor’s contract to the patient’s contract



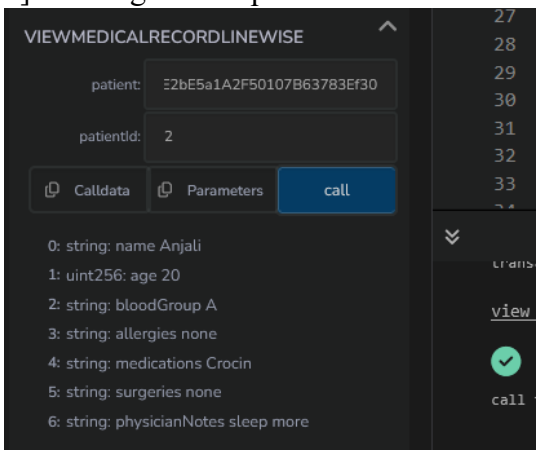
3] Authorizing the second doctor to the patient’s contract



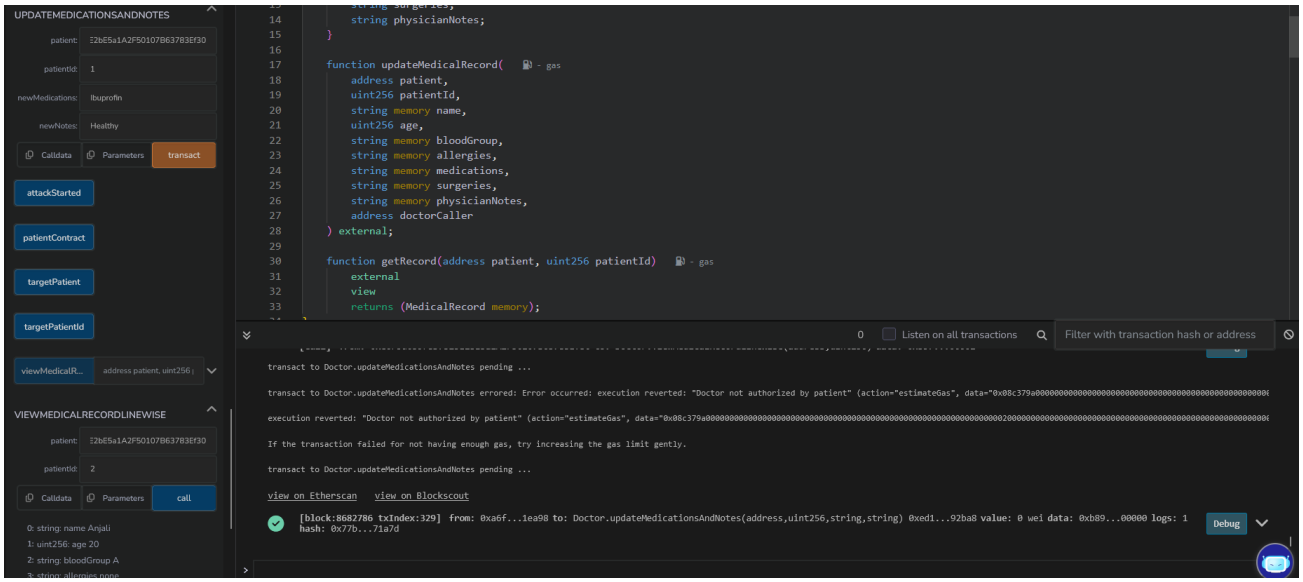
4] Viewing first patient's details in the second doctors contract



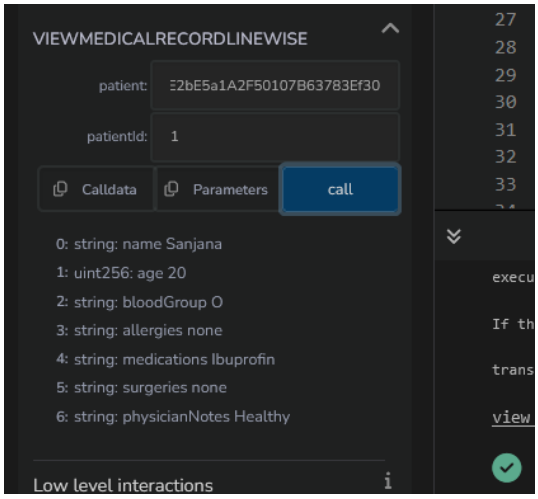
5] Viewing second patient's details in the second doctor's contract



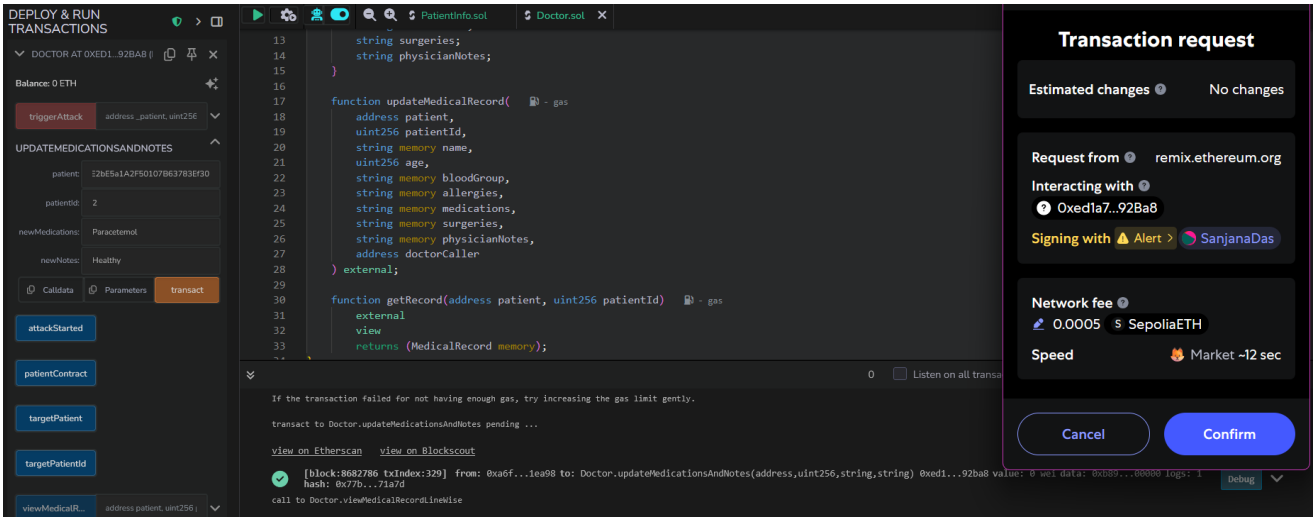
6] The second doctor updates the details of the first patient



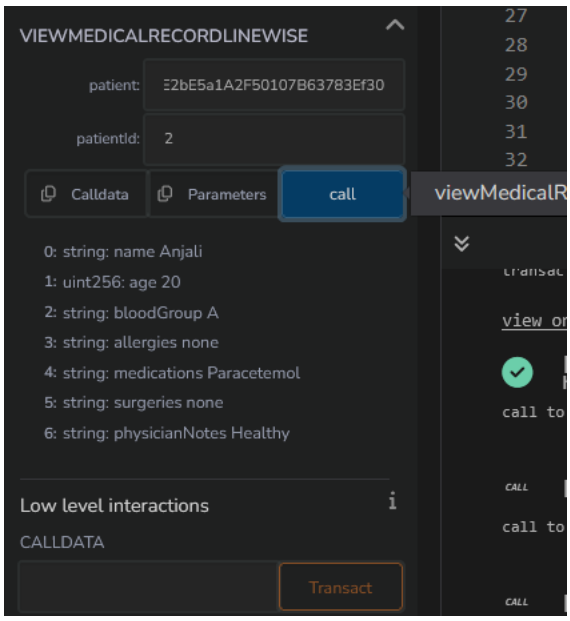
7] Viewing the updates made by the second doctor in the first patient's details



8] The second doctor updates the details of the second patient



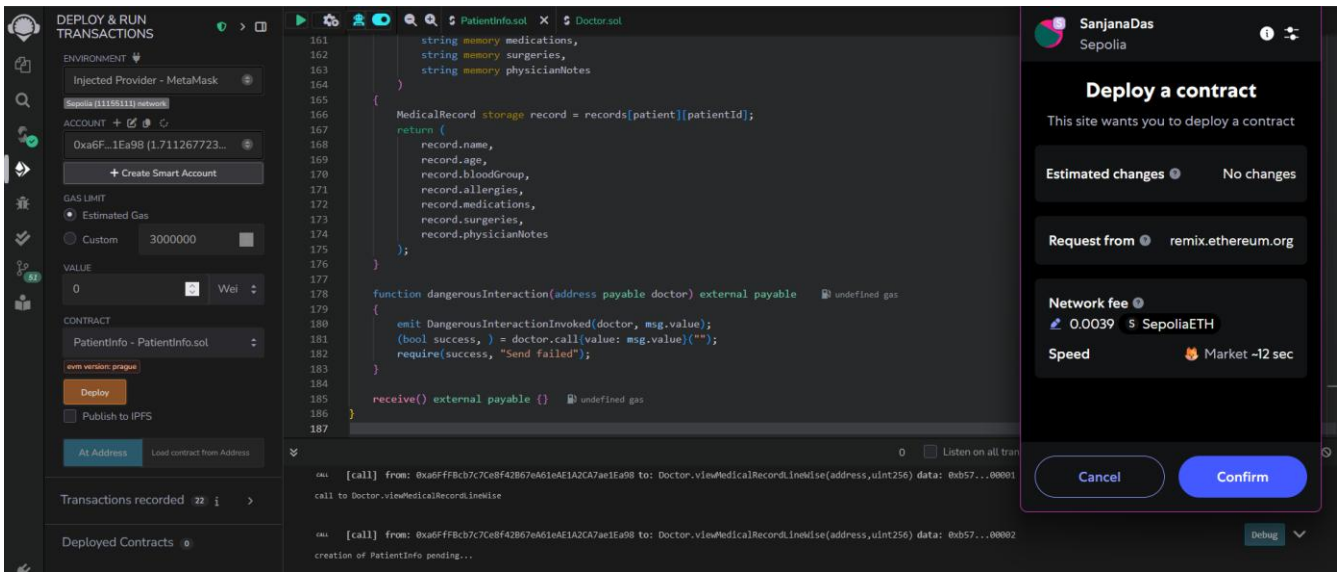
9] Viewing the updates made by the second doctor in the second patient's details



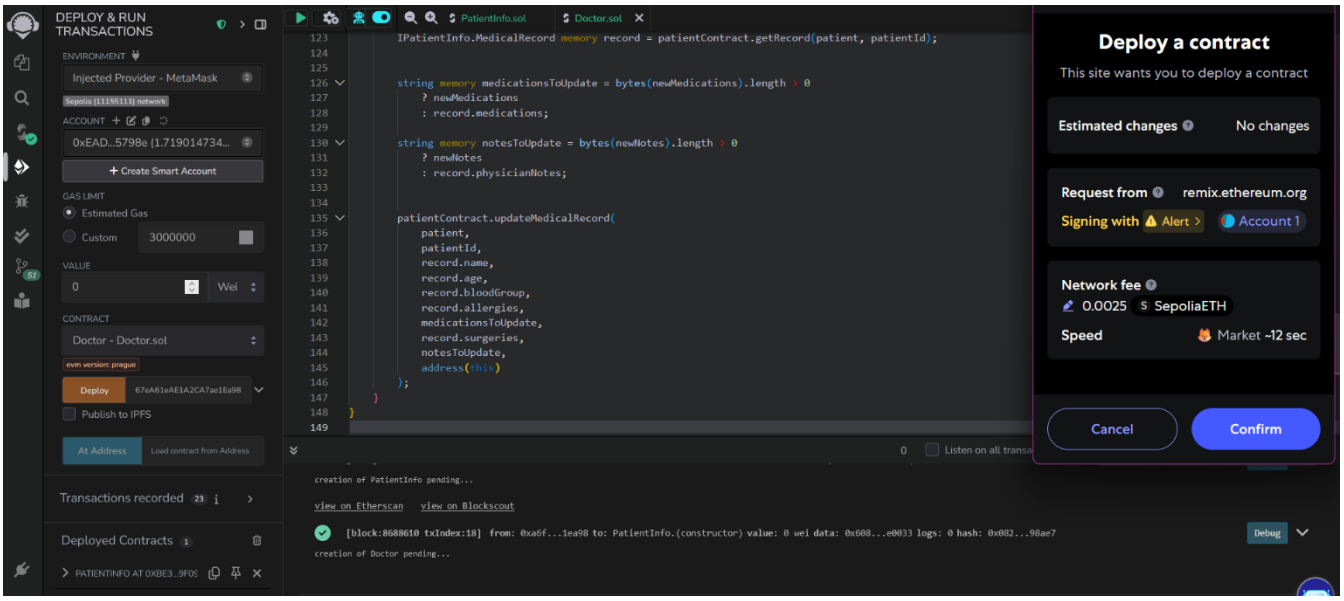
Behaviour of Re-enterancy attack with multiple doctors and multiple patients

1] Two doctor's one patient

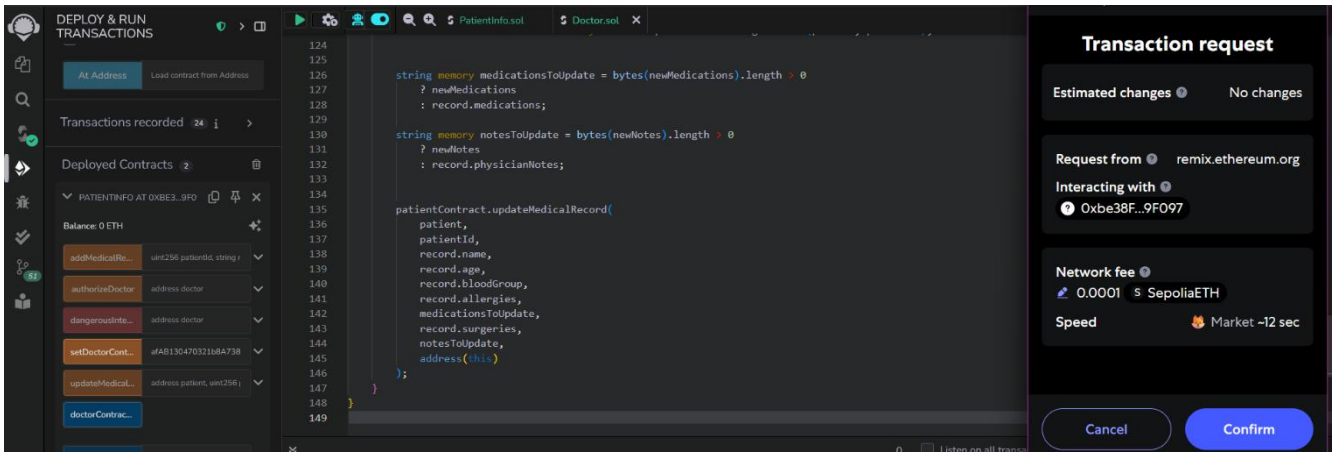
1] Deploying patient contract

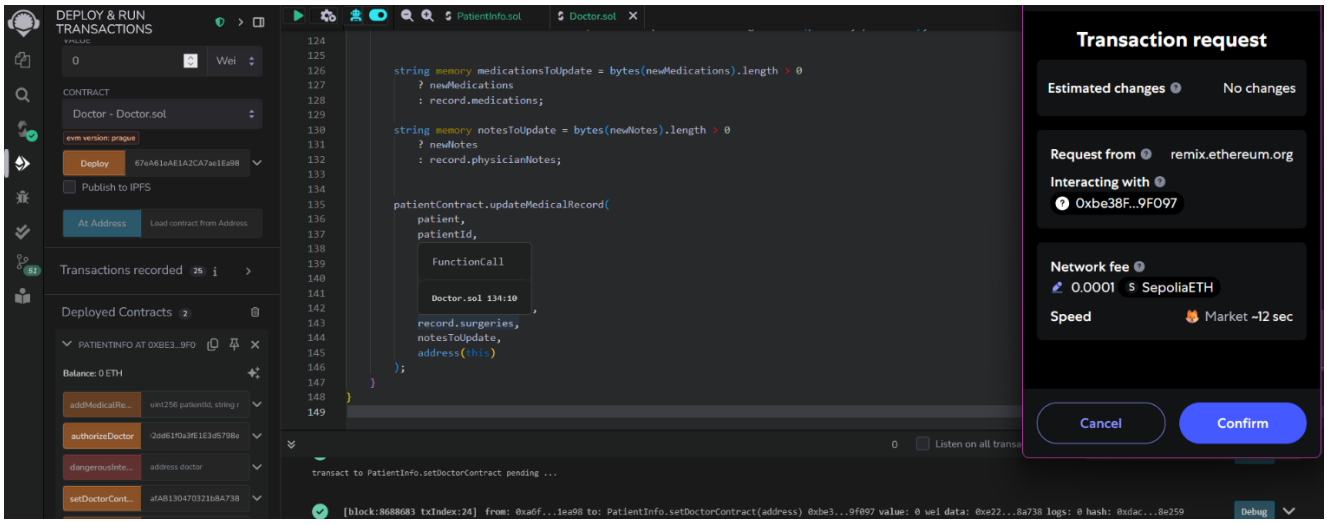


2] Deploying first doctor's contract

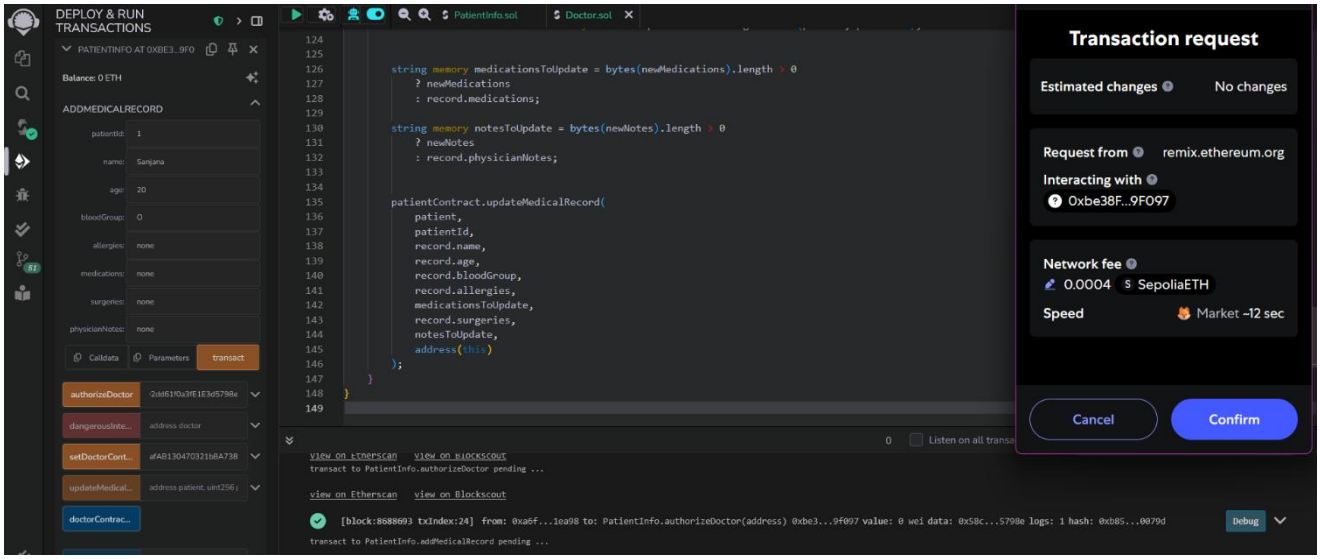


3] Setting and authorizing the first doctor within the patient's contract

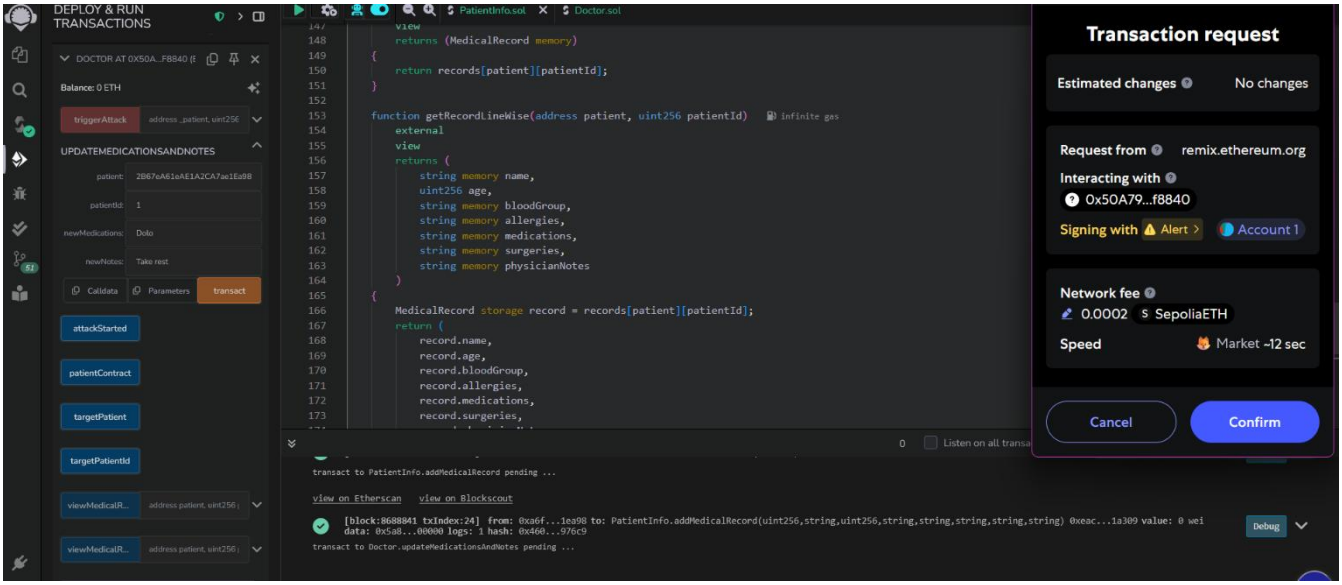




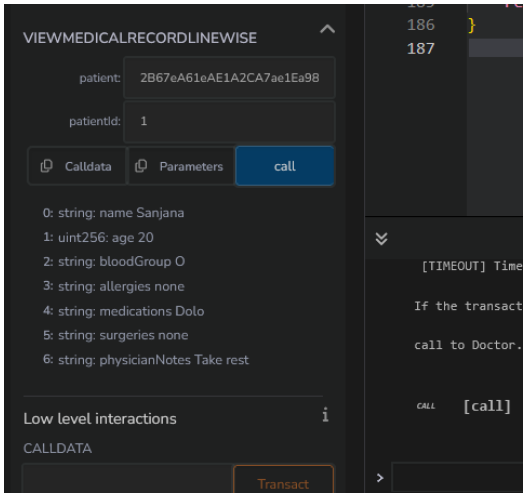
4] Adding patient's details



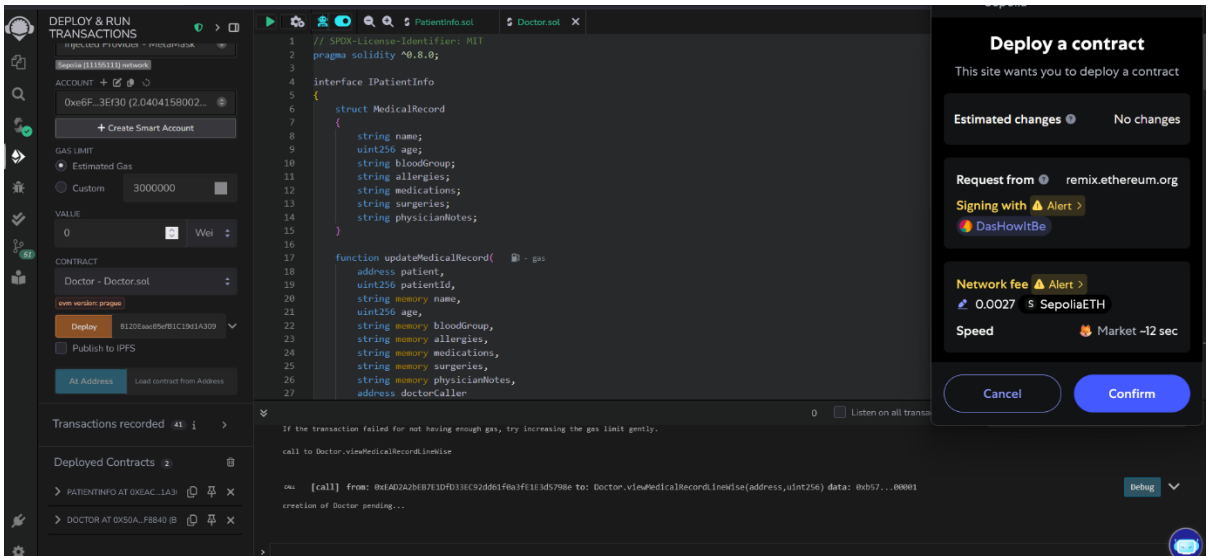
5] Updating patient's details on first doctor's contract



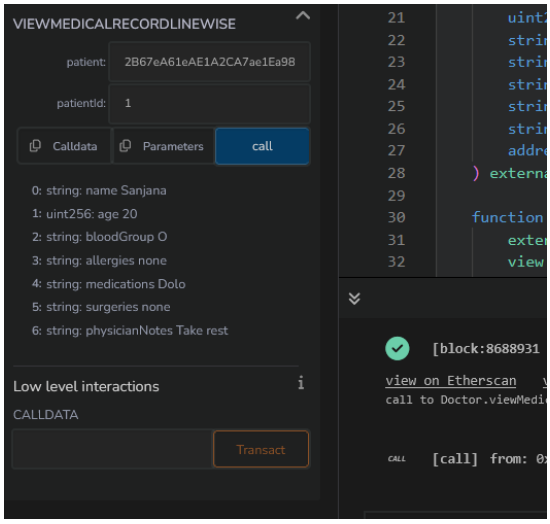
6] Viewing the updated patient's details



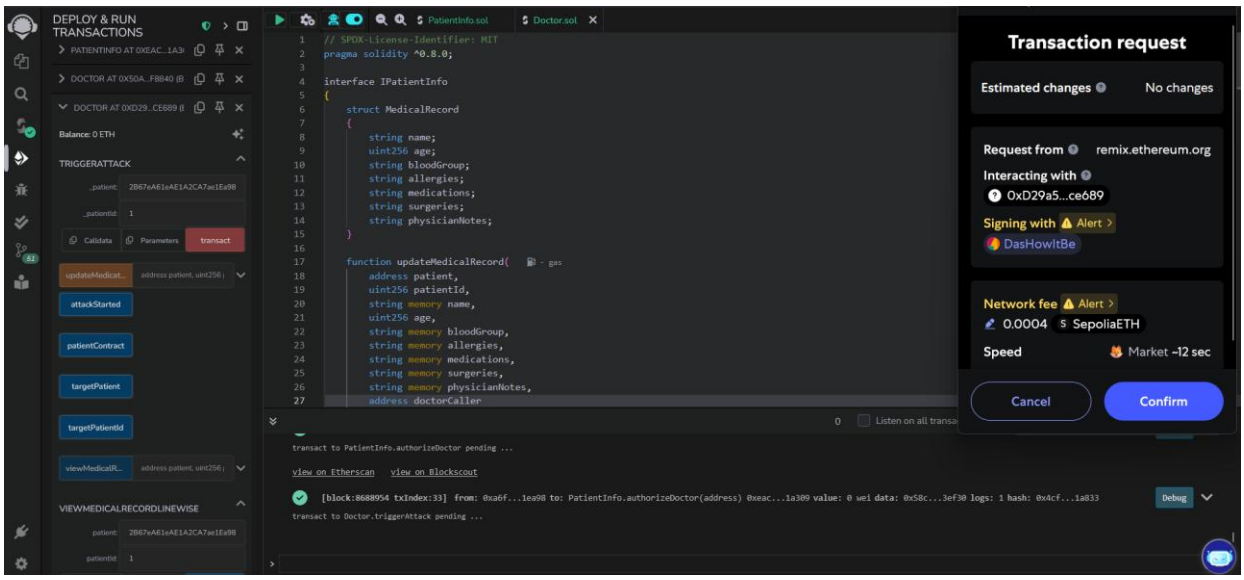
7] Deploying the second doctor's contract



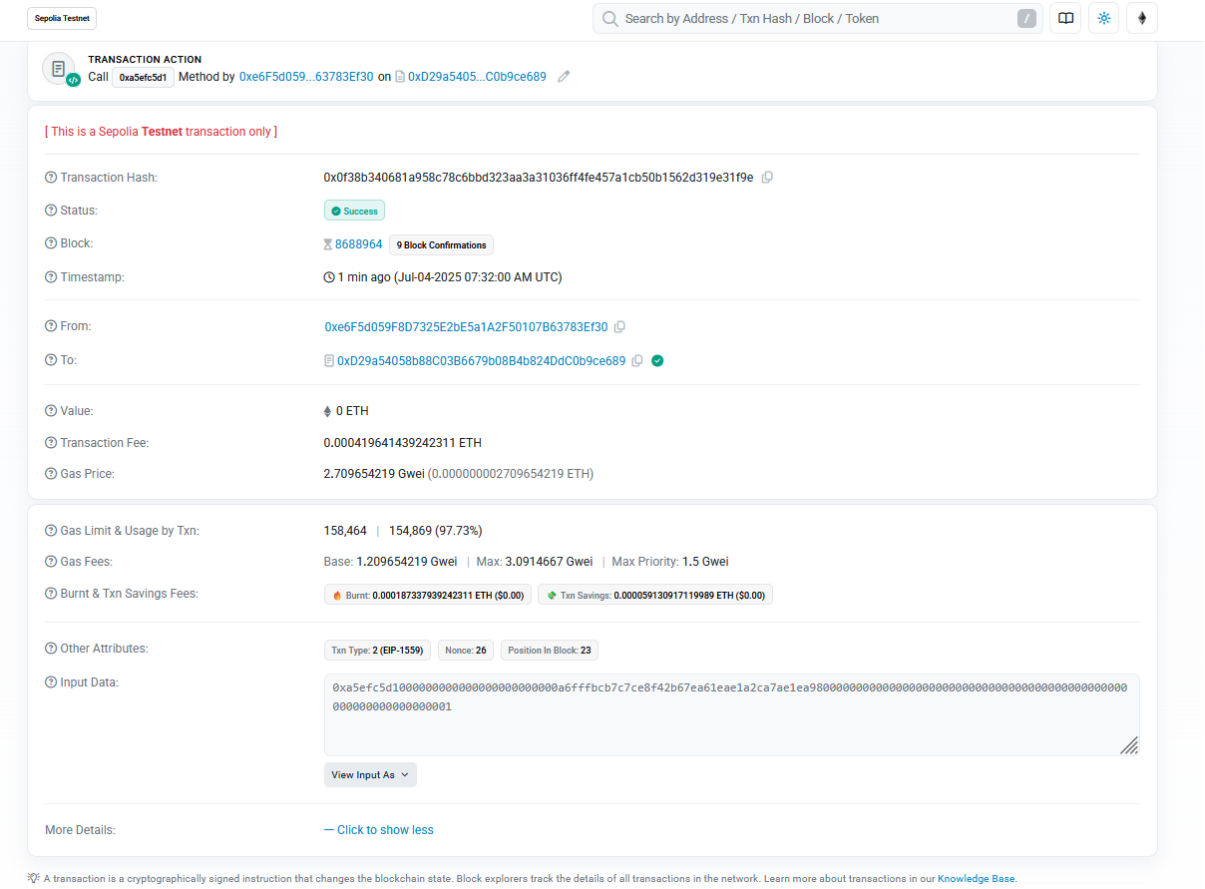
8] Viewing the patient's details in the second doctor's contract



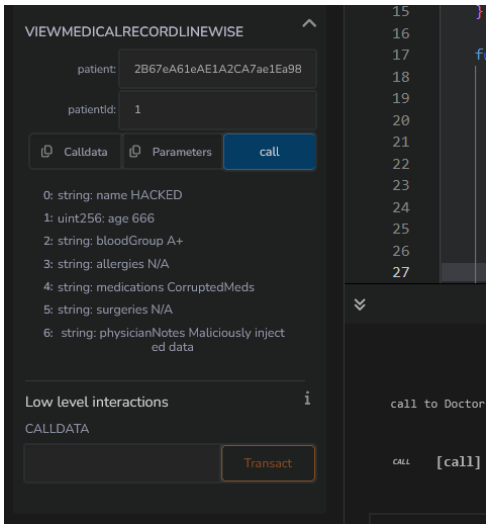
9] Now we are performing the reentrancy via the second doctor's contract



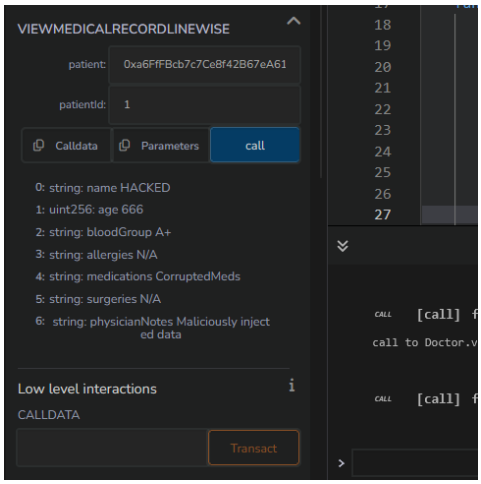
10] Viewing the attack on Etherscan



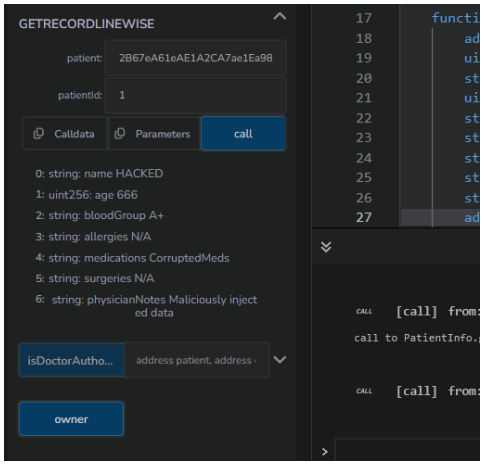
11] Viewing the malicious patient record after the reentrancy attack in the second doctor's contract



12] Viewing the malicious patient record after the reentrancy attack in the first doctor's contract

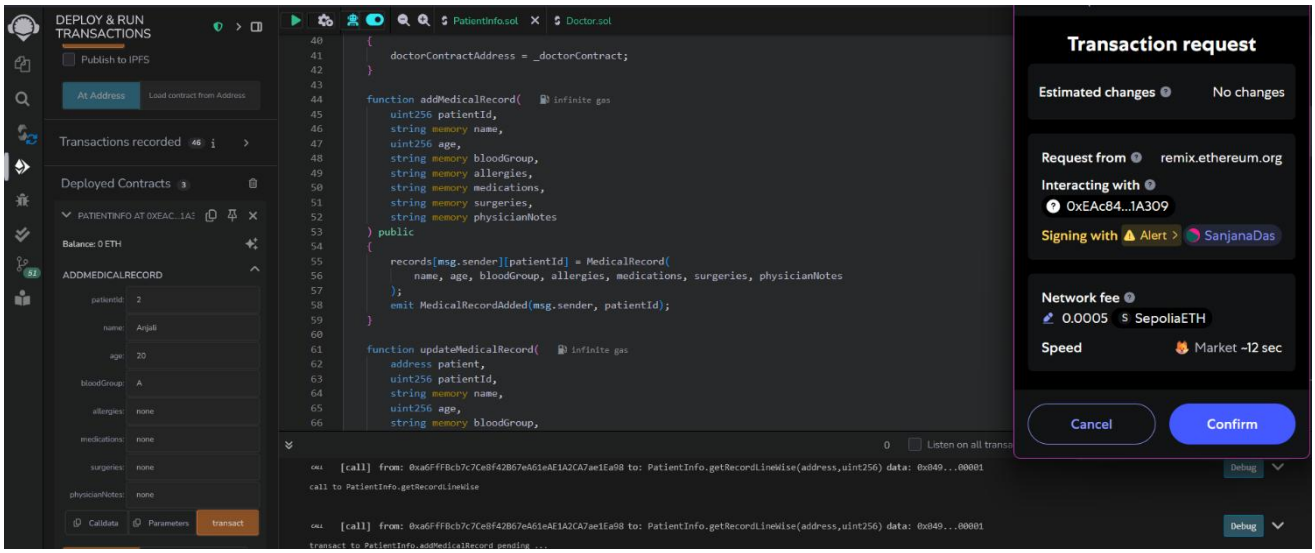


13] Viewing the malicious patient record after the reentrancy attack in the patient's contract

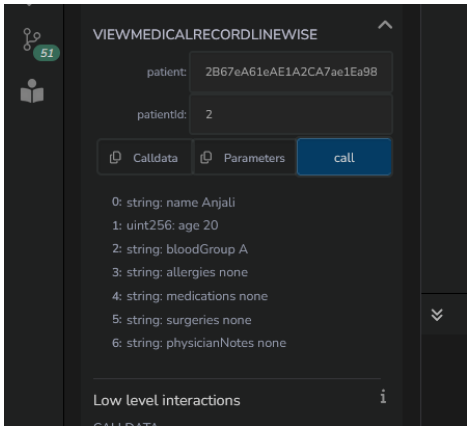


2] Checking condition with two patients

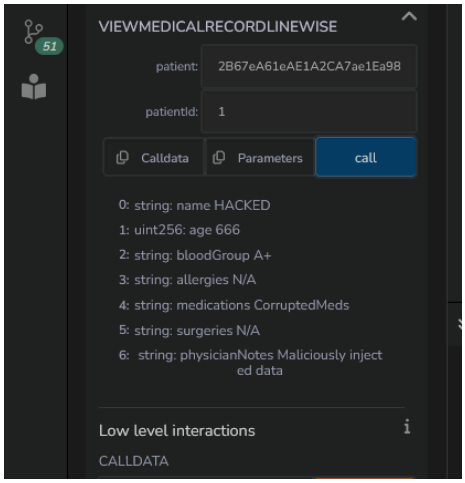
1] Adding another patient



2] Viewing second patient details



3] Viewing first patient details [the second patient's details does not affect the malicious data of the first patient



A Study on Re-Enterancy Vulnerability

Year	Website/Tool/Name	Method	Challenge Addressed
2020	Clairvoyance	Cross-function and cross-contract static analysis	Complex inter-contract data/control flow analysis
2020	Combined Static & Dynamic Analyzer Framework	Static and dynamic analysis, attacker contract generation using ABI	Accurate reentrancy detection in presence of complex behaviors
2020	Reguard, Contract Fuzzer	Fuzzing with ABI-guided attacker contract generation	Detecting runtime contract vulnerabilities like reentrancy
2021	Jyane	Path profiling solution	Detecting potential execution paths that may cause vulnerabilities
2021	Datalog-based Detection	Datalog-based formulation over EVM traces	Detecting real-world reentrant transactions
2022	CodeNet	Dataset-based analysis	Code-level vulnerability benchmarking
2022	Etherscan + CLOC Tool	API-based analysis with CLOC for LOC metrics	Code metrics and gas analysis for vulnerability detection
2022	Semantic Analysis (ML)	Machine learning-based semantic vulnerability detection	Efficient detection using learned code semantics
2022	Gas-Limited Transfer Mitigation	Use of transfer() method, mutex, update-before-call patterns	Preventing reentrancy via secure state and transfer patterns
2023	Image-based Detection	Deep learning on code images	Visual pattern recognition of code vulnerabilities
2023	Bi-GRU Attention Network	Multi-objective neural network using Bi-GRU and attention	Contextual vulnerability detection using sequence modeling

Year	Website/Tool/Name	Method	Challenge Addressed
2024	CNN & LSTM-Attention Model	Deep learning with CNN + LSTM + Attention	Learning long-range dependencies in smart contract logic
2024	Two-Phase Framework (Detection + Repair)	DL-based detection with attention-based repair via LLM prompts	Inadequate features, imbalanced datasets, poor domain-specific knowledge
2024	EvoFuzzer	Genetic algorithm optimized fuzzing with static pruning	Efficient test case generation for reentrancy attack simulation
2024	BlockWatchdog	Combines: Flow Extraction, Cross-contract Analysis, Attacker Detection	Inter-contract behavior understanding, attacker behavior modeling
2025	Contract Rule Framework	Static analysis with contract-standard rules	Early-stage rule-based detection before deployment

Detection Techniques:

1. Static Analysis:
    - Analyzes code without executing it.
    - Tools like Clairvoyance, BlockWatchdog, and static rule-based frameworks are used.
    - Efficient for identifying suspicious patterns across functions and contracts.
2. Dynamic Analysis:
    - Analyzes contract behavior during execution.
    - Tools like Reguard and attacker contract simulations are used to recreate real-world exploit scenarios.
    - Detects re-entrancy by mimicking actual attacks.
3. Fuzzing-based Detection:
    - Uses tools like Evofuzzer and contract fuzzers to generate inputs and observe responses.
    - Combines static analysis and genetic algorithms for test case evolution.
4. Machine Learning & Deep Learning:

    - CNN, LSTM, Bi-GRU-Attention, and multi-objective neural networks are applied for pattern detection and contextual learning.
    - These models improve precision and recall in detecting complex vulnerabilities.
5. Semantic and Symbolic Analysis:
    - Semantic analysis methods leverage the meaning of code for detection.
    - Datalog-based symbolic reasoning helps trace re-entrant calls through execution traces.

Mitigation Techniques:

1. Gas Limiting:
    - Restrict gas usage or hard-code gas limits to prevent recursive calls.
2. Transfer Methods:
    - Use transfer() instead of call.value() to limit the risk window.
3. State Variable Updates:
    - Always update contract state before external calls to prevent re-entry during unfinalized state.
4. Mutex and Access Controls:
    - Use reentrancy guards or mutex locks (like status variables) to block multiple entries into sensitive functions.
5. Contract Design Principles:
    - Use modular and standard-compliant contract design.
    - Follow the Checks-Effects-Interactions pattern for secure execution order.

Trends Over the Years:

- Early years (2020–2021): More focus on static/dynamic analysis and symbolic methods.
- 2022–2023: Introduction of deep learning models and semantic detection.
- 2024–2025: Advanced tools like LLM-based repair, cross-contract static analysis, and multi-objective ML.

USING TOOLS TO TEST VULNERABILITIES IN SMART CONTRACTS

1] Slither

What is Slither?

Slither is a powerful static analysis framework developed by Trail of Bits for Solidity smart contracts. It helps developers detect security vulnerabilities, optimize gas usage, and understand code structures. Slither performs:

- Static code analysis (no need to deploy)
- Detection of reentrancy, insecure external calls, tx.origin misuse
- Dataflow and taint analysis
- Code optimization suggestions
- Visualization of function call graphs, inheritance, and data dependencies

Why Use Slither in This Project?

The goal of this project is to secure medical records shared between a Patient contract and a Doctor contract. Using Slither ensures that:

- The code follows best practices
- There are no reentrancy risks
- All external call patterns are analyzed
- There are no critical or medium severity issues
- The logic of tx.origin usage is flagged (even if intentional)

Observations from Slither Analysis

PatientInfo.sol:

- No reentrancy vulnerabilities (use of nonReentrant is confirmed effective).
- Use of tx.origin detected: Slither flags this as a potential issue. In this context, it's used intentionally for identifying the transaction origin between linked contracts. It is handled securely, but this warning is expected.

- Low gas optimization suggestions (e.g., marking certain functions external instead of public).
- Proper access control modifiers (onlyOwner, isDoctorAuthorized) detected.
- No uninitialized storage or shadowed variables.

Doctor.sol:

- Purposeful fallback function with reentrancy is detected and flagged as high severity (intended for demonstration).
- Simulated attack vector is recognized but contained.
- No unsafe calls outside the reentrancy test scenario.
- Minor suggestion: Add emit statements where appropriate for logs.

2] Mythril

What is Mythril?

Mythril is an open-source security analysis tool for Ethereum smart contracts. It performs symbolic execution, taint analysis, and control flow checks to detect vulnerabilities like:

- Reentrancy
- Integer overflows
- Unchecked return values
- Tx.origin usage
- Uninitialized storage pointers

It supports analysis of contracts written in Solidity and deployed on EVM-compatible blockchains.

Why Use Mythril in This Project?

In this project, a Patient–Doctor smart contract system was developed to securely manage sensitive medical records. Mythril was used to:

- Analyze reentrancy attack surfaces
- Ensure the nonReentrant modifier is effective

- Validate removal of unsafe tx.origin usage patterns
- Detect unexpected vulnerabilities in update and payment functions

Observations from Mythril Scan

PatientInfo.sol:

- No reentrancy vulnerabilities detected after using the nonReentrant guard.
- No uninitialized storage variables.
- Use of tx.origin flagged: Mythril warns against relying on tx.origin for authorization. However, in this case, it's used intentionally and securely for cross-contract checks.
- Call to untrusted address (doctor.call) is now protected by nonReentrant.

Doctor.sol:

- Code structured for simulation of attack scenarios.
- Mythril confirms receive fallback function is a reentrancy entry point (intentionally designed for testing).
- Does not introduce unguarded call/invoke patterns.

Conclusion

The final implementation successfully mitigates the cross-contract reentrancy vulnerability by integrating a nonReentrant modifier using a status flag pattern in the PatientInfo contract. All critical functions that involve external calls or state changes are protected, particularly the dangerousInteraction and updateMedicalRecord functions. Additionally, authorization checks and clear access controls ensure that only verified doctors can update patient data. These security enhancements provide robust protection against reentrancy attacks, securing the integrity and confidentiality of sensitive medical records in a decentralized environment.

References

- https://www.youtube.com/watch?v=3T1t2ginfTg
- https://solidity-by-example.org/hacks/re-entrancy/
- https://www.youtube.com/watch?v=AYpftDFiIgk
- https://www.geeksforgeeks.org/solidity/smart-contracts-in-blockchain/
- https://moralis.com/how-to-write-a-smart-contract-in-solidity-full-guide/
- https://www.quicknode.com/guides/ethereum-development/smart-contracts/a-broad-overview-of-reentrancy-attacks-in-solidity-contracts
- medium.com/@olatunjimayowa0396/the-reentrancy-attack-patterns-for-solidity-smart-contract-e0bf90e46b17
- https://www.nadcab.com/blog/reentrancy-guard-in-smart-contract
- https://www.udemy.com/course/master-ethereum-and-solidity-programming-with-real-world-apps/learn/lecture/26460714#overview
- https://www.youtube.com/watch?v=YtNvhcI3mac
- https://www.youtube.com/watch?v=YNZ-3l4S4M0
- https://youtu.be/fhOMTYkw8FM?si=ls836pig2zAMsb0K
- https://youtu.be/YNZ-3l4S4M0?si=b90t3ewMzNom8Usn
- https://youtu.be/YtNvhcI3mac?si=oNBFYNgSVPH6DXI4
- https://youtu.be/Dxex3b-eMq0?si=L4iMozJWdsar-Yyd
- https://youtu.be/jJUNMXKO_nw?si=8MRR6Qqg7k4gDii6