

Lab
Access Control and Security Issues
in Smart Contracts:
Healthcare Scenario

Version 1.1

LERSAIS

School of Computing and Information
University of Pittsburg

Goal

The goal of this lab is to illustrate how to manage access control and security issues of using Ethereum smart contract in healthcare scenarios.

In this lab, you will learn the following objects:

- 1) The access control in smart contracts
- 2) The common security issues in smart contracts and their countermeasures

Part 0 Required Tools

This lab requires the following tools:

- 1) Get *Node* and *npm* installed.
 - a. <https://www.npmjs.com/get-npm>
 - b. If you are not sure whether your system installed or not, you can check the following commands in the terminal environment.

```
$ node -v
```

```
$ npm -v
```

If the commands return the corresponding version number, you are all set; otherwise, follow the instruction from the above link to install the tools.
- 2) Get the following packages installed
 - a. Install 'ethereumjs-util' package through <https://github.com/ethereumjs/ethereumjs-util>
 - b. Install 'web3-utils' package through <https://github.com/ethereum/web3.js/tree/1.0/packages/web3-utils>
- 3) You will use the following online tools
 - a. Remix (<https://remix.ethereum.org/>)
 - b. Etherscan (<https://etherscan.io/>)
 - c. MetaMask (<https://metamask.io/>)

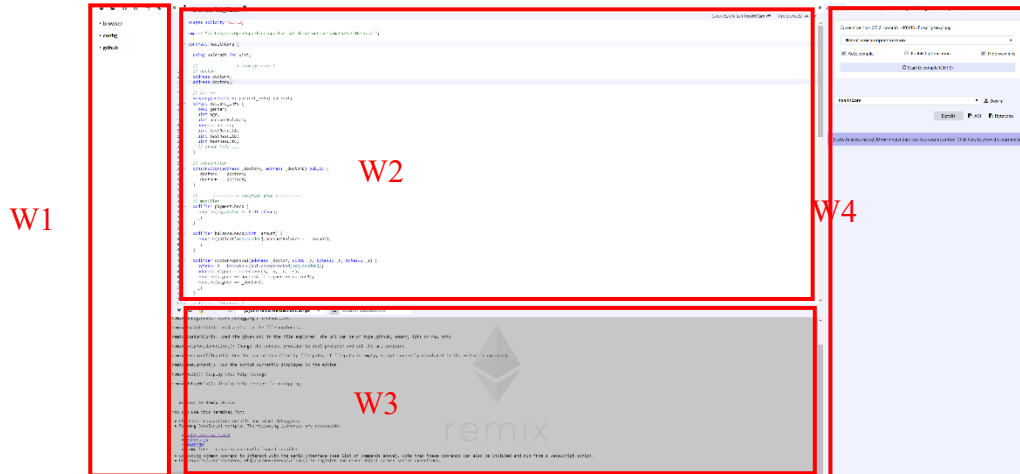
Note that the screenshot in the rest of the lab is captured in the Win10 environment. The node and npm tool also support other operation system.

Part I Environment setup

In this part, we describe the environment setup.

1) Open **Chrome** browser, go to **Remix** (<https://remix.ethereum.org/>).

a. You will see a webpage like this:



b. **Remix** is an online Ethereum IDE that consists of four windows:

- W1: storage browser
- W2: editor
- W3: console
- W4: control panel

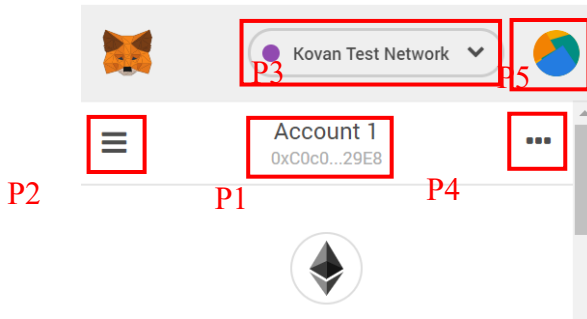
c. The rest of this lab extensively relies on the use of Remix. For more details, please refer to the following documents:

- <https://theethereum.wiki/w/index.php/Remix>
- <https://remix.readthedocs.io/en/latest/>

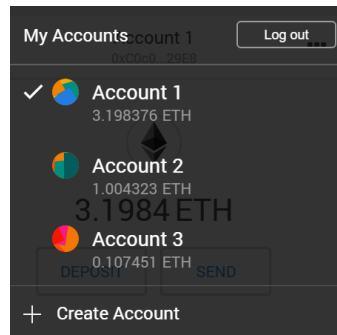
2) Install **MetaMask** (<https://metamask.io/>) as an extension of **Chrome** browser.



- You will see the extension icon of **MetaMask** on your **Chrome** browser, click it.
- Show a screenshot of **MetaMask** after the click.
- Create an account.
- Good job! You now own a **MetaMask** account as shown below. This is your first account and is named Account 1. In this lab, you will need to set three accounts. Before going further, please first get familiar with this **MetaMask** control panel.



- P1: you can copy the address of account 1 by clicking it.
 - P2: you can get more details of account 1.
 - P3: you can switch Ethereum networks.
 - P4: some more functionalities.
 - P5: you can manage your accounts.
- e. Please click P3 and show a screenshot of the optional networks. Please make sure that you are at the *Kovan* Test Network throughout this lab.
- f. Now, let's create two more accounts. Click P5 and then 'create account'. Please create two accounts and call them Account 2 and Account 3, respectively. You should now have three accounts like this:



- g. What are account balances of your three accounts?
- h. What are the addresses of your three accounts?

3) Next, let's get some free 'money' for your accounts.

- a. Go to the *Kovan faucet* (<https://faucet.kovan.network/>).
- b. Login with your *Github* account. If you don't have one, please create a new *Github* account for free.

- c. You will then see:

This faucet allows to request KETH once every 24h.

Kovan address..

Send me KETH!

- d. Input your Account 1 address and click ‘send me KETH’. You should then see:

This faucet allows to request KETH once every 24h.

KETH on it way! See the transaction on [Etherscan](#)

- e. Click ‘**Etherscan**’, you will then be navigated to <https://kovan.etherscan.io/> where you can monitor the **Kovan** Ethereum blockchain. You will see the details of the transaction that the faucet account sent 1 Ether to your Account 1. **Show a screenshot of the transaction. What is the address of the faucet account?**
- f. At the above ‘**Etherscan**’ page showing transaction details, click your account 1 address after ‘To:’. You will be navigated to your account page. **Have you received the 1 Ether? Show a screenshot of your account balance.**
- g. Good job! Your Account 1 now should have 1 Ether. Next, let’s transfer 0.2 Ether from Account 1 to Account 2 and Account 3, respectively.
- Click P4 of MetaMask control panel and then click ‘Expand View’. You should then see a new window opened in **Chrome**. This is an expanded version of MetaMask control panel. In the rest of this lab, we will only use this expanded version.
 - You should see this (your balance should be 1 ETH):



- iii. Click ‘SEND’, you should see this:

Send ETH ×

Only send ETH to an Ethereum address.

From:




To:

Amount: Max No Conversion Rate Available

Transaction Fee:

Slow	Average	Fast
0.00008 ETH	0.00021 ETH	0.00042 ETH

[Advanced Options](#)

- iv. Change the ‘Recipient Address’ to your Account 2 address and change the value of ‘Amount’ from 0 ETH to 0.2 ETH. Click ‘NEXT’ and later ‘CONFIRM’.
 - v. Redo iv. for your Account 3. **Now what are the balances of your three accounts?**
- 4) Go back to **Remix**. In W1, click + of    and name the new file as ‘lab_01.sol’ and click ‘OK’. You should then find this new in browser folder of W1. Open it to W2 and copy the contract codes below (in blue) to W2.
 - 5) Good job! You have completes all tasks of this part. In the next part, we will study access control of smart contrasts using the **healthCare** contract you have just copied.

```

pragma solidity ^0.5.2;
import "github.com/OpenZeppelin/zeppelin-solidity/contracts/math/SafeMath.sol";
contract healthCare {
    using SafeMath for uint;

    // -----> storage area <-----
    // doctor
    address doctorA;
    address doctorB;

    // patient
    mapping(address => patient_info) patient;
    struct patient_info {
        bool gender;
        uint age;
        uint accountBalance;
        address doctor;
        uint testResultA;
        uint testResultB;
        uint testResultC;
        // other info ...
    }

    // constructor
    constructor(address _doctorA, address _doctorB) public {
        doctorA = _doctorA;
        doctorB = _doctorB;
    }

    // -----> modifier area <-----
    // modifier
    modifier paymentCheck {
        require(msg.value >= 0.01 ether);
        _;
    }

    modifier balanceCheck(uint _amount) {
        require(patient[msg.sender].accountBalance >= _amount);
        _;
    }

    modifier doctorApproval(address _doctor, uint8 _v, bytes32 _r, bytes32 _s) {
        bytes32 h = keccak256(abi.encodePacked(msg.sender));
        address signer = ecrecover(h, _v, _r, _s);
        require(signer == doctorA || signer == doctorB);
        require(signer == _doctor);
        _;
    }
}

```

```

modifier allDoctors {
  require(msg.sender == doctorA || msg.sender == doctorB);
}

modifier yourDoctorOnly(address _patient) {
  require(msg.sender == patient[_patient].doctor);
}

// -----> function area <-----

// register
function patientRegister(bool _gender, uint _age, address _doctor, uint256 _v, bytes32 _r, bytes32 _s) public payable paymentCheck
doctorApproval(_doctor, uint8(_v), _r, _s) {
  patient[msg.sender].gender = _gender;
  patient[msg.sender].age = _age;
  patient[msg.sender].accountBalance = msg.value - 0.01 ether;
  patient[msg.sender].doctor = _doctor;
}

// payment
function recharge() payable public {
  patient[msg.sender].accountBalance += msg.value;
}

function withdraw_danger(uint _amount) public balanceCheck(_amount) {
  msg.sender.call.value(_amount)("");
  patient[msg.sender].accountBalance -= _amount;
}

function withdraw_safe(uint _amount) public balanceCheck(_amount) {
  patient[msg.sender].accountBalance -= _amount;
  msg.sender.transfer(_amount);
}

// setter
function setTestResultA(address _patient, uint _testResultA) public allDoctors {
  patient[_patient].testResultA = _testResultA;
}

function setTestResultB(address _patient, uint _testResultB) public yourDoctorOnly(_patient) {
  patient[_patient].testResultB = _testResultB;
}

function setTestResultC_danger(address _patient) public allDoctors {
  patient[_patient].testResultC = patient[_patient].testResultA + patient[_patient].testResultB;
}

function setTestResultC_safe(address _patient) public allDoctors {
  patient[_patient].testResultC = patient[_patient].testResultA.add(patient[_patient].testResultB);
}

// getter
function getPatientInfo(address _patient) public view returns(
  bool gender,
  uint age,
  uint accountBalance,
  address doctor,
  uint testResultA,
  uint testResultB,
  uint testResultC) {
  gender = patient[_patient].gender;
  age = patient[_patient].age;
  accountBalance = patient[_patient].accountBalance;
  doctor = patient[_patient].doctor;
  testResultA = patient[_patient].testResultA;
  testResultB = patient[_patient].testResultB;
  testResultC = patient[_patient].testResultC;
  // other info ...
}

// other functions ...
}

```

Part II Access Control in Smart Contracts

In this part, you will learn access control in smart contracts using your three accounts and the *healthcare* contract.

- 1) First, let's get more familiar with the *healthcare* contract.
 - a. In the storage area of this contract, we record the addresses of doctor A and doctor B and also use a mapping and a struct to record information of each patient. Specifically, in this lab, *please use your Account 1 as the patient account, Account 2 as the doctor A account and Account 3 as the doctor B account.*
 - b. We would like this contract to be created and controlled by two doctors (A and B), indicating that either doctor A or B should deploy the contract to the Ethereum network.
 - c. After that, a patient can call the function *patientRegister()* to make his or her information get recorded. To be able to call this function, ***the patient must pay at least 0.01 ETH and submit the approval of doctor A or doctor B (access control 1).***
 - d. Later, the test results A, B, C can be set by doctors. ***Both doctor A or B can set TestResultA or TestResultC while only the doctor providing approval to the patient can set TestResultB for the patient (access control 2).***

2) Now, let's implement 1) a. and b.

- a. Switch to your Account 2 in *MetaMask*, which corresponds to doctor A.
- b. In *Remix* W4, select compiler version 0.5.2:

Current version:0.5.2+commit.1df8f40c.Emscripten.clang

Select new compiler version ▼

Auto compile

Enable Optimization

Hide warnings

- c. Compile 'lab_01.sol'. You should see this in *Remix* W4. Ignore the warnings.

The screenshot shows the Remix IDE interface. At the top, there is a dropdown menu with 'healthCare' selected and a 'Swarm' icon. Below the dropdown are three buttons: 'Details', 'ABI', and 'Bytecode'. At the bottom, there is a purple warning banner that reads: 'Static Analysis raised 30 warning(s) that requires your attention. Click here to show the warning(s). ✖'

d. Switch from ‘compile’ to ‘run’ in **Remix** W4. You will see this:

The screenshot shows the Remix IDE interface with the 'Run' tab selected. The interface is divided into three main sections, each highlighted with a red box and labeled P1, P2, and P3.

P1: Environment settings. It shows 'Injected Web3' as the environment, 'Kovan (42)' as the network, and 'Account 2' (0xc0c...a29e8) as the selected account. Other settings include 'Gas limit' (3000000) and 'Value' (0 wei).

P2: Deployment options. It shows the 'healthCare' contract selected. There are two options: 'Deploy' (with a red button) and 'At Address' (with a blue button). The 'Deploy' option has a dropdown menu showing 'address_doctorA, address_doctorB'.

P3: Deployed Contracts. It shows a section titled 'Deployed Contracts' with a trash icon. Below it, it says 'Currently you have no contract instances to interact with.'

- P1: Make sure your environment is **Kovan**. If not, change it in **MetaMask**. Your current account may be Account 2. You may change your account in **MetaMask**.
- P2: you will deploy smart contracts here.
- P3: you will call functions of deployed smart contract here.

e. Now let's use your Account 2 (i.e., doctor A) to deploy this **healthcare** contract.

- Expand ‘deploy’ in P2. You should see:

The screenshot shows the 'Deploy' form in the Remix IDE. It has two input fields for '_doctorA:' and '_doctorB:', each with 'address' as a placeholder. A 'transact' button is visible at the bottom right.

- Input your Account 2 address and Account 3 address as the addresses for doctor A and B, respectively. Then click transact.
- You will see a **MetaMask** notification window, click ‘confirm’.
- **Have you seen any change in P3? Please expand the new deployed contract and show a screenshot.**

- f. Good job! You have just deployed your first smart contract, the *healthcare* contract.
- 3) Next, let's implement 1). c, namely the first access control problem. In P3 of the Remix W4. You have expanded the deployed healthcare contract and have seen all the functions. Let's first register a new patient.
- a. Switch to Account 1 in *MetaMask*. Make sure that the account displayed in P1 of *Remix* W4 is now your Account 1 (i.e., patient).
 - b. Expand function *patientRegister()* in P3, input 'true', '30' to `_gender` and `_age` and also input your Account 2 address to `_doctor`.
 - c. Click 'transact' of function *patientRegister()*, **are you seeing a *MetaMask* notification window? If not, show what's new in *Remix* W3.**
 - d. The reason that you failed to call the function *patientRegister()* is that you did not meet the two access control policies. By checking the codes of this function in *Remix* W2, you may find that this function has two *Modifiers*, namely *paymentCheck* and *doctorApproval*.
 - i. The *paymentCheck Modifier* requires that the patient to pay at least 0.01ETH to be registered.
 - ii. The *doctorApproval Modifier* requires the patient to get the approval from doctor A or B.
 - e. To satisfy the *paymentCheck* access control policy, you may easily change the value in P1 of *Remix* W4 from '0 wei' to '0.01 ether'. After that, by the time you call the function *patientRegister()*, your transaction will also send 0.01 ETH to the healthcare contract.
 - f. To satisfy the *doctorApproval* access control policy, you will need to get a signature from doctor A or B, say doctor A in this example. Now, imagine that you are doctor A (i.e., your Account 2) and you will generate the signature for a patient (i.e., your Account 1). You need to do the following:
 - i. Open your terminal (i.e., command prompt in windows)
 - ii. `$ node`
 - iii. `>var ethUtils = require('ethereumjs-util');`
 - iv. `>var web3Utils = require('web3-utils');`
 - v. `>var addr_bytes = new Buffer('Account1Addr', 'hex')`

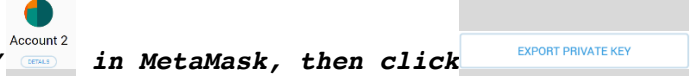
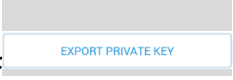
1. Please replace `AccountAddr` with `abs...e23` if your Account 1 address is `0xabs...e23`

vi. `>var addr_hash = ethUtils.keccak256(addr_bytes)`

vii. `>var h_code = '0x'+ addr_hash.toString('hex')`

viii. `>var privkey = new Buffer('Account2Privkey', 'hex')`

1. Please replace `Account2Privkey` with the private key of your Account 2. You may find the key by click

 'Details' in MetaMask, then click , input your password and finally click 'Confirm'. **What is the private key?**

ix. `>var h_code_hex = new Buffer(h_code.slice(2), 'hex');`

x. `>var vrs = ethUtils.ecsign(h_code_hex, privkey);`

xi. `>var v = vrs.v`

xii. `>var r = vrs.r.toString('hex')`

xiii. `>var s = vrs.s.toString('hex')`

- g. Please list the `v`, `r`, `s`. They three together form the signature, namely the approval that doctor A will assign to the patient.
- h. Okay, now you can input the `v,r,s` to function `patientRegister()` in P3 of **Remix** W4. Please double-check that you have changed the value in P1 of **Remix** W4 to '0.01 ether'. Also, please switch to the patient account (i.e., your Account 1).
- i. Now, click 'transact' of function `patientRegister()`. You should see a popup **Metamask** notification window, click 'Confirm'. You will see a new link in W3 of **Remix** W4. By clicking the link, you will be navigated to the transaction you just sent in **Etherscan**. Please show a screenshot of your transaction as well as a screenshot of the **healthcare** contract in **Etherscan**. What is the current balance of the **healthcare** contract?
- j. Finally, input the patient address (i.e., your Account 1) to the function `getPatienInfo()` in P3 of **Remix** W4 and call this function. Please list the current information of the patient.
- 4) Next, let's implement 1). d, namely the second access control problem.
- a. The function `setTestResultA()` allows setting the test result A. It uses the **allDoctors Modifier**, so both Doctor A and B can pass its access control policy. Please keep using your Account 1 (i.e., patient), expand the function

setTestResultA(), input your Account 1 address and 115792089237316195423570985008687907853269984665640564039457584007913129639934 as `_patient` and `_testResultA` respectively. **By clicking ‘transact’, can you successfully send out this transaction? What is the problem?**

- b. Now switch your account to Account 3 (i.e., doctor B), redo 4).a, **did the transaction get sent out this time? Redo 3).j, what is the current information of the patient?**
 - c. The function *setTestResultB()* allows setting the test result B. Doctor B (i.e., your Account 3) wants to set the test result B as 1 through calling this function. **Will he be successful? Way?**
 - d. Switch your account to Account 2 (i.e, doctor A), redo 4).c, **will doctor A be successful? What is the current information of the patient?**
- 5) Good job! You have completed the study of access control in smart contracts. Let’s go to the last part of this lab!

Part III Hierarchical Role Based Access Control

In this part, you will go on a relatively complex access control smart contract comparing to the last part. code that we are going to use is given below. You need to paste it into the remix web browser. Write the file name as **blockchain_lab3_<yourname>.sol**

```
pragma solidity ^0.5.1;
import "github.com/OpenZeppelin/zeppelin-solidity/contracts/math/SafeMath.sol";

contract roleBasedAccessControl
{
    using SafeMath for uint;
    // -----> storage area <-----

    // First Role: Doctor
    address doctor;

    // Second Role: Nurse
    mapping(address => nurse_info) nurse;
    struct nurse_info
    {
        bool gender;
        uint age;
        uint accountBalance;
        address doctor;
    }

    // Third Role: Patient
    mapping(address => patient_info) patient;
    struct patient_info
    {
        bool gender;
        uint age;
        uint accountBalance;
        //address doctor;
        address nurse;
    }
}
```

```

}

// constructor
constructor(address _doctor) public
{
    doctor = _doctor;
}
    // -----> modifier area <-----

// modifier
modifier paymentCheck
{
    require(msg.value >= 0.01 ether);
}
_

modifier balanceCheck(uint _amount)
{
    require(patient[msg.sender].accountBalance >= _amount);
}
_

modifier doctorApproval(address _doctor, uint8 _v, bytes32 _r, bytes32 _s)
{
    bytes32 h = keccak256(abi.encodePacked(msg.sender));
    address signer = ecrecover(h, _v, _r, _s);
    //require(signer == doctor);
    require(signer == _doctor);
}
_

modifier nurseApproval(address _nurse, uint8 _v, bytes32 _r, bytes32 _s)
{
    bytes32 h = keccak256(abi.encodePacked(msg.sender));
    address signer = ecrecover(h, _v, _r, _s);
    //require(signer == nurse);
    require(signer == _nurse);
}
_

    // -----> function area <-----

// doctor approves the nurse to get register in the system
function nurseRegister(bool _gender, uint _age, address _doctor, uint256 _v, bytes32 _r, bytes32 _s) public payable paymentCheck
doctorApproval(_doctor, uint8(_v), _r, _s)
{
    nurse[msg.sender].gender = _gender;
    nurse[msg.sender].age = _age;
    nurse[msg.sender].accountBalance = msg.value - 0.01 ether;
    nurse[msg.sender].doctor = _doctor;
}

// nurse approves the patient to get approved into the system
function patientRegister(bool _gender, uint _age, address _nurse, uint256 _v, bytes32 _r, bytes32 _s) public payable paymentCheck
nurseApproval(_nurse, uint8(_v), _r, _s)
{
    patient[msg.sender].gender = _gender;
    patient[msg.sender].age = _age;
    patient[msg.sender].accountBalance = msg.value - 0.01 ether;
    patient[msg.sender].nurse = _nurse;
}

// payment
function recharge() payable public
{
    patient[msg.sender].accountBalance += msg.value;
}

function withdraw_danger(uint _amount) public balanceCheck(_amount)
{
    msg.sender.call.value(_amount)("");
    patient[msg.sender].accountBalance -= _amount;
}

function withdraw_safe(uint _amount) public balanceCheck(_amount)
{

```

```

    patient[msg.sender].accountBalance -= _amount;
    msg.sender.transfer(_amount);
}

// getter function for the nurse's information
function getNurseInfo(address _nurse) public view returns
(
    bool gender,
    uint age,
    uint accountBalance,
    address doctor
)
{
    gender = nurse[_nurse].gender;
    age = nurse[_nurse].age;
    accountBalance = nurse[_nurse].accountBalance;
    doctor = nurse[_nurse].doctor;
}

// getter function for patient's information
function getPatientInfo(address _patient) public view returns
(
    bool gender,
    uint age,
    uint accountBalance,
    address nurse
)
{
    gender = patient[_patient].gender;
    age = patient[_patient].age;
    accountBalance = patient[_patient].accountBalance;
    nurse = patient[_patient].nurse;
}
}

```

This code demonstrates one of the examples of hierarchical role-based access control in the healthcare sector. This concepts and code logic can be extended/used for any application which requires hierarchy. Hierarchy that is used is doctor approves nurse, nurse approves patients into the system.

Remember, this one is the advanced lab on Blockchain and you need to build on top of what you already learnt on prior two labs. So, it requires step by step solution from your end. Code on which you need to build up is provided. You can use that and need to include and modify it where ever applicable.

List of tasks you need to perform is given below-

1. Can you provide any other hierarchical role-based example? List 3 such examples.
2. In lab-2, you already acquire some hands-on experience on two types of access control. How you can accomplish that here? Write step by step approach that you performed to observe the hierarchical role-based access control (code is already provided). Steps should be detail and you are supposed to provide screenshots at every step.
3. You need to change the code-sample to include the following scenario:
New hierarchy should have **hospitalOwner->doctor->headNurse->Nurse->{Patient and one family member of patient}** i.e. either of the patient or one of his close family member can access the system on behalf of the patient. This facility is needed so that when

the patient is unable to provide access or in dying situation, his/her family members can provide the necessary authorization. Submit the code and output. Name this code as **newHiaerarchy_yourname.sol**

4. Can you add 5 doctors into the doctor's list instead of just one? Remember this is your new code. So, don't change into the previous one directly. Rather create another version and change on that. Name this code as **fiveDoctors_yourName.sol**
5. Submit both the code and include screenshots for every step.
6. In all the labs, extensively we are using mapping and modifier concepts. In your own words, write down what you need that. Is there any alternative by which the same applicability can be shown but without using modifier or mapping? If so, write down the alternative code snippet.
7. With all the labs, you gained some fare enough idea on several blockchain based application in healthcare scenario. Now, can you write think about a new application irrespective of all which you already did? There is no need to write the code. Instead, demonstrate it using diagram.
8. In the **fiveDoctors_yourName.sol** smart contract, can you point out at least three security vulnerability? [Hint: you can refer to lab-2 manual]

Submit all the mentioned codes and screenshots.

Congratulations!! You have successfully completed all the blockchain labs.

Part IV Common Security Issues in Smart Contracts and Countermeasures

In this last part, you will learn two common security issues of smart contracts and their corresponding countermeasures. Let's first study the *integer-overflow* problem and then the *reentrancy* attack.

- 1) The *integer-overflow* problem:

- a. In part II.4), we have set testResultA and testResultB to 115792089237316195423570985008687907853269984665640564039457584007913129639934 and 1, respectively.
 - b. Now, use your Account 2 (i.e., doctor A) to call function `setTestResultC_danger()` with the patient address. **What is the current information of the patient? What is the current test result C of this patient?**
 - c. Redo part II.4).d by using your Account 2 to set test result B of the patient to 2. Then, redo part III.1).b. **What is the current test result C of this patient?**
 - d. **Can you explain the reason?** Hint: In Solidity, the uint256 data type supports integer in the range $[0, 2^{256}-1]$.
 - e. The *integer-overflow* problem can be notified by using the *SafeMath* library, which have been imported in the *healthcare* contract.
 - f. Now, use your Account 2 (i.e., doctor A) to call function `setTestResultC_safe()` with the patient address. **Can you successfully send out the transaction? Show the screenshot of the popup window.**
 - g. [optional] **Why the *SafeMath* library can help? Please refer to the function `add()` in the 'SafeMath.sol' file.** You may find it under the *github* folder in *Remix* W1.
- 2) The c:
- a. Create a new smart contract in *Remix* and call it 'lab_02.sol'.
 - b. Copy the following codes into 'lab_02.sol'.

```
pragma solidity ^0.5.2;

contract attack {

    address payable owner;
    address target;

    constructor(address _target) public payable {
        owner = msg.sender;
        target = _target;
    }

    function register(bool _gender, uint _age, address _doctor, uint8 _v, bytes32 _r, bytes32 _s) public {
        target.call.value(0.02 ether)(abi.encodeWithSignature("patientRegister(bool,uint256,address,uint256,bytes32,bytes32)",
        _gender, _age, _doctor, _v, _r, _s));
    }

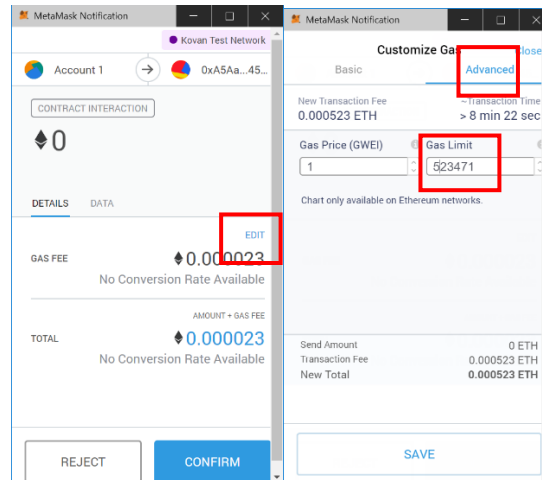
    function callTarget() public {
        target.call(abi.encodeWithSignature("withdraw_danger(uint256)", 0.01 ether));
    }

    function() payable external {
        target.call(abi.encodeWithSignature("withdraw_danger(uint256)", 0.01 ether));
    }

    function withdraw() public {
        selfdestruct(owner);
    }
}
```


}

- c. This is the **attack** contract. In Ethereum, transactions may be sent by either a External Owned Account (EOA) controlled by a user through a pair of keys or a Contract Account (CA) controlled by a deployed smart contract. In this scenario, you will perform as an adversary, who deploys this attack contract and use the deployed attack contract to launch the **reentrancy** attack for the purpose of stealing money in the **healthcare** contract.
- d. First, use your Account 2 to transfer 0.1 ETH to the **healthcare** contract. You can do this by setting the value to '0.1 ether' and call the function `recharge()` of the **healthcare** contract. **Show the screenshot of the current balance of the **healthcare** contract in *Etherscan*.**
- e. Now, you may use your Account 1 as the adversary to perform the the **healthcare** contract. To do that, switch to Account 1, compile the **attack** contract and deploy it with the address of the **healthcare** contract as argument. Please make sure that you set value in P1 of Remix W4 to '0.02 ether' before you deploy the **attack** contract, this will give the deployed **attack** contract a balance of 0.02 ether, which will be used later to launch the **reentrancy** attack. **Show a screenshot of the **attack** contract in *Etherscan*.**
- f. The **attack** contract has four functions. The first task of the adversary is to register the **attack** contract as a patient in the **healthcare** contract. To do that, use your Account 1 to call the function `register()`. The arguments can follow the ones in part II.3).f. However, the signature `v,r,s` needs to be recomputed by changing the **Account1Addr** in part II.3).f.v to the address of the **attack** contract. (i.e., if the **attack** contract address is `0xfds...3gf`, then use `fds...3gf`). **Call the `getPatienInfo()` in P3 of *Remix W4* with the **attack** contract address, show the result.**
 - i. Notice: You will need to increase the gas limit. Specifically, when you see the popup MetaMask notification window, click 'EDIT', then switch to 'Advanced' and set 'Gas Limit' to a value over 500,000.



- g. Next, use your Account 1 to call the function `callTarget()` in the *attack* contract. This time, please increase the gas limit to 3,000,000. Show the screenshot of the balance of the *attack* contract and the *healthcare* contract in Etherscan.
- h. You have successfully launched the *reentrancy* attack. The function `callTarget()` has called the `withdraw_danger()` function in the *healthcare* contract, requesting the *healthcare* contract to refund 0.01 ETH to the *attack* contract. Remember that the *attack* contract owned 0.02 ETH at the beginning. It registered itself to the *healthcare* contract with this 0.02 ETH, so its balance in the *healthcare* contract is 0.01 ETH. Upon being called, the `withdraw_danger()` function first check whether the balance of the *attack* contract in the *healthcare* contract has at least 0.01 ETH. If the result is true, the `withdraw_danger()` function will send 0.01 ETH back to the *attack* contract and decrease the balance of the *attack* contract by 0.01 ETH. Unfortunately, when the `withdraw_danger()` function execute sthe code `'msg.sender.call.value(_amount)("");'` to transfer 0.01 ETH to the *attack* contract, it will automatically invoke the fallback function (i.e., the function with no name) in the *attack* contract. The *attack* contract leverage this fallback function to reenter the *healthcare* contract by calling the `withdraw_danger()` function again. At this moment, the code `'patient[msg.sender].accountBalance -= _amount;'` has not been executed, so the balance of the *attack* contract in the *healthcare* contract is still 0.01 ETH and another 0.01 ETH will be sent back to the *attack* contract, which will invoke the fallback function in the *attack* contract again...

- i. There are two ways to resolve the *reentrancy* attack. One is to replace the low-level *call* function in `'msg.sender.call.value(_amount)("");'` to a *transfer* function. **What is the other way?** Hint: compare *withdraw_danger()* and *withdraw_safe()*.
 - j. Finally, use your Account 1 to call the function *withdraw()* in the *attack* contract. **Show the screenshot of the balance of the *attack* contract and your Account 1.**
- 3) Good job! You have completed this lab! Congratulations!