

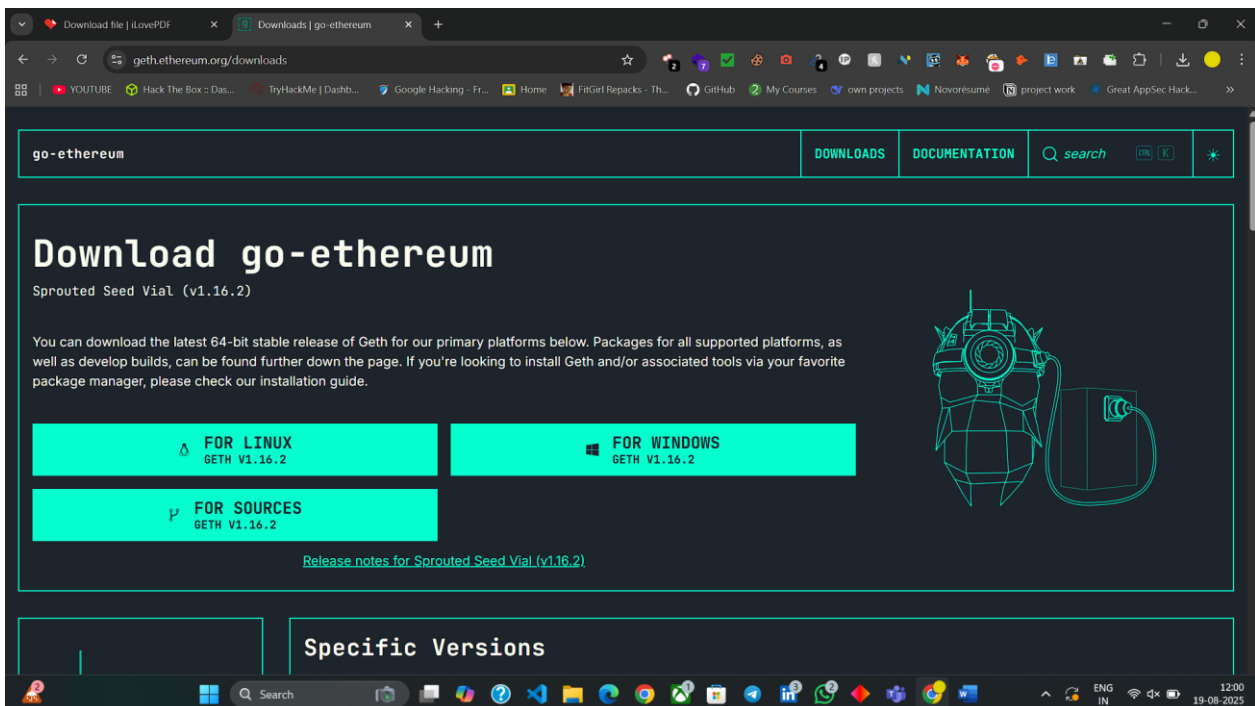
## Practical No - 1

### AIM: Introduction to Block Chain

**Description:** This practical focuses on understanding the concept of blockchain and implementing a private Ethereum blockchain network using Geth (Go Ethereum). A blockchain is a distributed digital ledger that stores transactions in blocks linked in sequential order. In this activity, instead of connecting to the public Ethereum mainnet, we build a private network where only authorized nodes can participate.

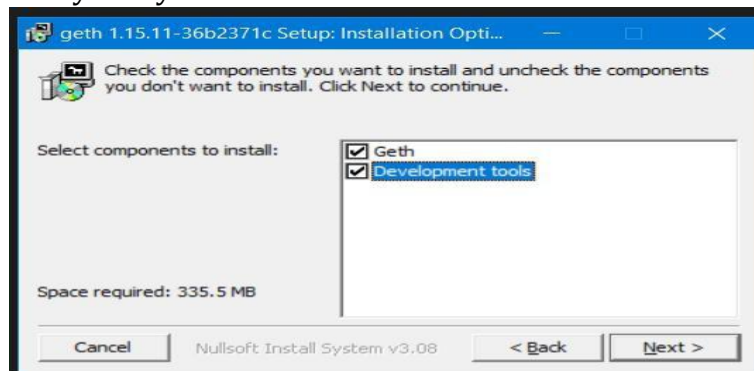
### Step-01:

- Now install Ethereum in your system to build a private blockchain using it



### Step-02:

- Now install Ethereum in your system



- Add its path to the environment variables

**Step-03:**

- Now create the folder name “eth private” by using this floder we wil start to build or private blockchain.
- Now open the terminal and give command “cd <eth private folder location >
- Now in that directory give “geth” to check the geth is running on the system

```

C:\Users\yash2\Ethprivate>geth --version
geth version 1.16.1-stable-12b4131f

C:\Users\yash2\Ethprivate>geth
INFO [08-19]12:03:10.762] Starting Geth on Ethereum mainnet...
INFO [08-19]12:03:10.764] Bumping default cache on mainnet
INFO [08-19]12:03:10.765] Maximum peer count
WARN [08-19]12:03:10.772] Sanitizing cache to Go's GC limits
INFO [08-19]12:03:10.772] Set global gas cap
INFO [08-19]12:03:10.772] Initializing the KZG library
INFO [08-19]12:03:10.772] Allocated trie memory caches
INFO [08-19]12:03:10.778] Using pebble as the backing database
INFO [08-19]12:03:10.778] Allocated cache and file handles
eum\geth\chaindata cache=1.22GiB handles=8192
INFO [08-19]12:03:10.820] Opened ancient database
eum\geth\chaindata\ancient\chain readonly=false
INFO [08-19]12:03:10.820] Opened Era store
um\geth\chaindata\ancient\chain\era
INFO [08-19]12:03:10.820] State scheme set to already existing
INFO [08-19]12:03:10.824] Initialising Ethereum protocol
WARN [08-19]12:03:10.824] Sanitizing invalid node buffer size
INFO [08-19]12:03:10.843] Opened ancient database
eum\geth\chaindata\ancient\state readonly=false
INFO [08-19]12:03:10.845] Initialized path database
INFO [08-19]12:03:10.845] Initialized path database
ffer=256.00MiB history="last 90000 blocks"

provided=1024 updated=4096
ETH=50 total=50
provided=4096 updated=2507
cap=50,000,000
backend=gokzg
clean=376.00MiB dirty=626.00MiB

database=C:\Users\yash2\AppData\Local\Ether
database=C:\Users\yash2\AppData\Local\Ether
datadir=C:\Users\yash2\AppData\Local\Ethere
scheme=path
network=1 dbversion=9
provided=626.00MiB updated=256.00MiB
database=C:\Users\yash2\AppData\Local\Ether
triecache=376.00MiB statecache=250.00MiB bu

```

**Step-04:**

- Now open vs code and develop a code to build genesis block(First block) of the private block chain in json format

```

1 {
2   "config": {
3     "chainId": 987,
4     "homesteadBlock": 0,
5     "eip150Block": 0,
6     "eip155Block": 0,
7     "eip158Block": 0,
8     "byzantiumBlock": 0,
9     "constantinopleBlock": 0,
10    "petersburgBlock": 0,
11    "istanbulBlock": 0,
12    "terminalTotalDifficulty": 0
13  },
14  "difficulty": "0x400",
15  "gaslimit": "0x8000000",
16  "alloc": {}
17 }

```

**Step-05:**

- Now open the terminal in vscode and change the directory into “eth private”

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS Z:\Sem 7 Files\Class work\BCWS\ethereumPersonal>

```

**Step-06:**

- Now execute the genesis file by giving command " **geth --datadir "C:\Users\sarik\ethprivate" init "C:\Users\sarik\Ethprivate\genesis.json"**
- By executing this command we successfully write the genesis block in our blockchain

```

3      "chainId": 987,
4      "homesteadBlock": 0,
5      "eip150Block": 0,
6      "eip155Block": 0,
7      "eip158Block": 0,
8      "byzantiumBlock": 0,
9      "constantinopleBlock": 0,
10     "petersburgBlock": 0,
11     "istanbulBlock": 0,
12     "terminalTotalDifficulty": 0
13   },
14   "difficulty": "0x400",
15   "gasLimit": "0x8000000",
16   "alloc": {}

```

PS Z:\Sem 7 Files\Class work\BCWS\ethereumPersonal> **geth --datadir "Z:\Sem 7 Files\Class work\BCWS\ethereumPersonal" init "Z:\Sem 7 Files\Class work\BCWS\ethereumPersonal\genesis.json"**

INFO [08-18|18:38:07.358] Maximum peer count ETH=50 total=50  
 INFO [08-18|18:38:07.386] Set global gas cap cap=50,000,000  
 INFO [08-18|18:38:07.389] Initializing the KZG library backend=gokzg  
 INFO [08-18|18:38:07.403] Defaulting to pebble as the backing database  
 INFO [08-18|18:38:07.404] Allocated cache and file handles database="Z:\Sem 7 Files\Class work\BCWS\ethereumPersonal\geth\chaindata" cache=16.00MiB handles=16  
 INFO [08-18|18:38:07.594] Opened ancient database database="Z:\Sem 7 Files\Class work\BCWS\ethereumPersonal\geth\chaindata\ancient" chain="main" gendata=false

**Step-07:**

- Initialize the private network Launch the private network in which various nodes can add new blocks for this we have to run the command" **geth --datadir "C:\User\sarik\ethprivate" --networkid 987 --http --http.addr "127.0.0.1" --http.port 8545 --http.api personal,eth,net,web3,miner --port 30303 console"**

```

6      "eip155Block": 0,
7      "eip158Block": 0,
8      "byzantiumBlock": 0,
9      "constantinopleBlock": 0,
10     "petersburgBlock": 0,
11     "istanbulBlock": 0,
12     "terminalTotalDifficulty": 0
13   },
14   "difficulty": "0x400",
15   "gasLimit": "0x8000000",
16   "alloc": {}

```

INFO [08-18|18:38:07.697] Writing custom genesis block  
 Fatal: Failed to write genesis block: unsupported fork ordering: eip158Block not enabled, but byzantiumBlock enabled at block 0

PS Z:\Sem 7 Files\Class work\BCWS\ethereumPersonal> **geth --datadir "Z:\Sem 7 Files\Class work\BCWS\ethereumPersonal" --networkid 987 --http --http.addr "127.0.0.1" --http.port 8545 --http.api pegeth --datadir "Z:\Sem 7 Files\Class work\BCWS\ethereumPersonal" --networkid 987 --http --http.addr "127.0.0.1" --http.port 8545 --http.api personal,eth,next,web3,miner --port 30303 console**

INFO [08-18|18:41:04.916] Maximum peer count ETH=50 total=50  
 INFO [08-18|18:41:04.930] Set global gas cap cap=50,000,000  
 INFO [08-18|18:41:04.930] Initializing the KZG library backend=gokzg  
 INFO [08-18|18:41:04.934] Allocated trie memory caches clean=154.00MiB dirty=256.00MiB  
 INFO [08-18|18:41:04.934] Using pebble as the backing database  
 INFO [08-18|18:41:04.935] Allocated cache and file handles database="Z:\Sem 7 Files\Class work\BCWS\ethereumPersonal\geth\chaindata" c

- After the we will get Geth javascript console where we can monitor how many peer are available in network and how many have connected to our private blockchain network it will show the peercount

**Conclusion:**

By performing this practical we learn the concept of blockchain and we learnt create genesis block which is first block in our private blockchain network.

## Practical No - 2

### Aim: Smart Contracts Development:

1. Understanding smart contracts.
2. Developing and deploying a basic smart contract.
3. Interacting with smart contracts.

### 1. Understanding smart contracts.

**Description:** Smart contracts have gained significant attention and popularity in the world of blockchain and cryptocurrency. Ethereum, with its robust blockchain infrastructure, has emerged as a leading platform for deploying dApps and executing smart contracts. While Web2 developers may be accustomed to building centralized applications, transitioning to Web3 development on Ethereum opens up a world of decentralized possibilities.

#### Prerequisites:

- Before diving into the process of deploying a smart contract on Ethereum, it is essential to have a basic understanding of the following concepts and meet the requirements:
- Familiarity with the fundamentals of blockchain technology (distributed ledger, consensus mechanisms, immutability).
- Key differences between centralized and decentralized applications (decentralized consensus, peer-to-peer networks, significance of smart contracts).
- Knowledge of Solidity, the programming language used to write smart contracts on the Ethereum Blockchain.
- Familiarity with blockchain terminologies (wallets, gas, block explorers, etc.).
- Ensure that you have Metamask browser extension using this link set up on your PC. If you haven't done so already, you can add the Metamask to your preferred web browser.
- Get the Sepolia testnet faucet here to enable you to cover the gas fees for deploying your smart contract on the Ethereum testnet.
- A text editor: For this tutorial, we will make use of REMIX, a development environment that allows you to write and test smart contracts. It provides a user friendly interface and a range of features to facilitate the development and deployment of smart contracts.

## Decentralized applications (DApps)

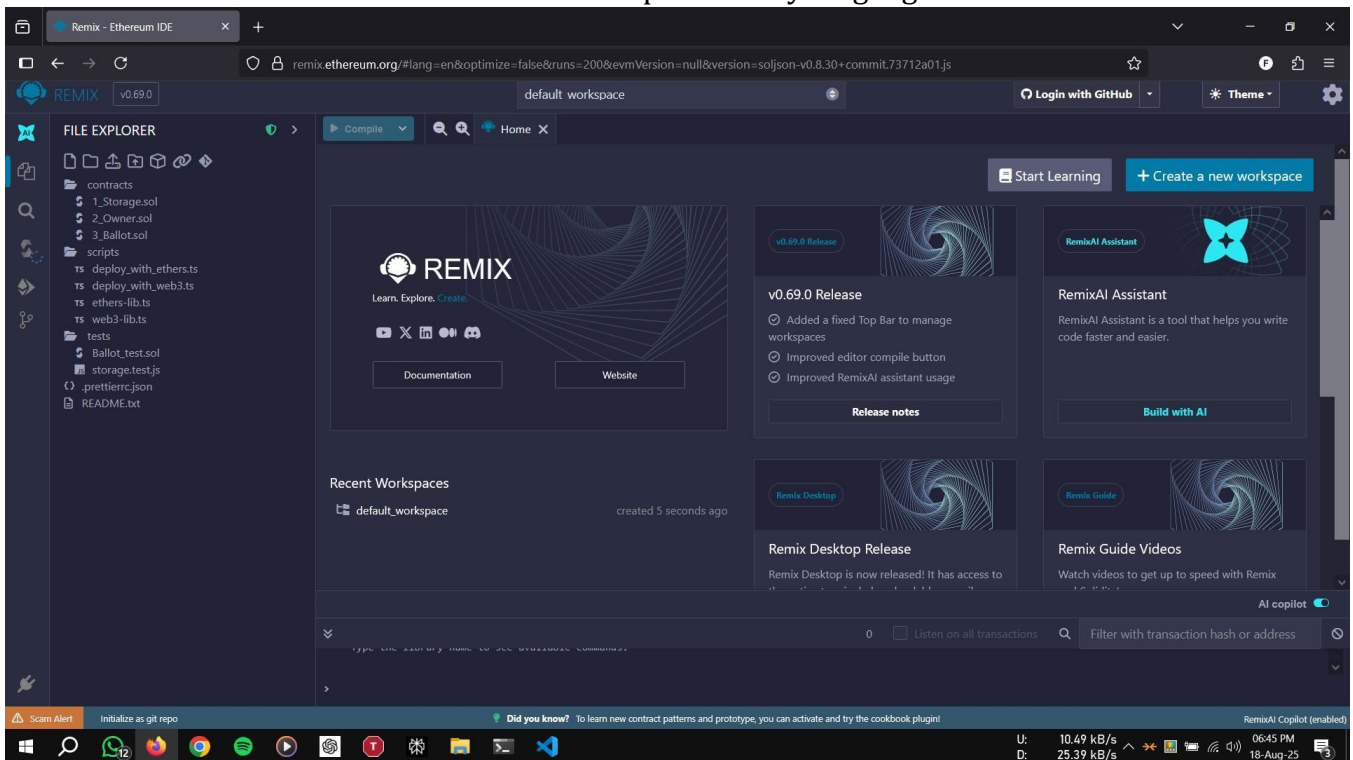
A Dapp is an application that operates in a decentralized manner. It utilizes smart contracts to store the application's business logic and state, replacing the traditional server-side component. However, it's important to note that DApps encompass more than just smart contracts. At the minimum, a DApp consists of:

- Smart contracts on a blockchain, and
- A web user interface. BCWS LABORATORY 303105416 2203031260056 Page | 8 In a broader sense, a Dapp is a web application that leverages open, decentralized, peer-to-peer infrastructure services.

## 2.Developing and Deploying smart

### Contracts Step-01:

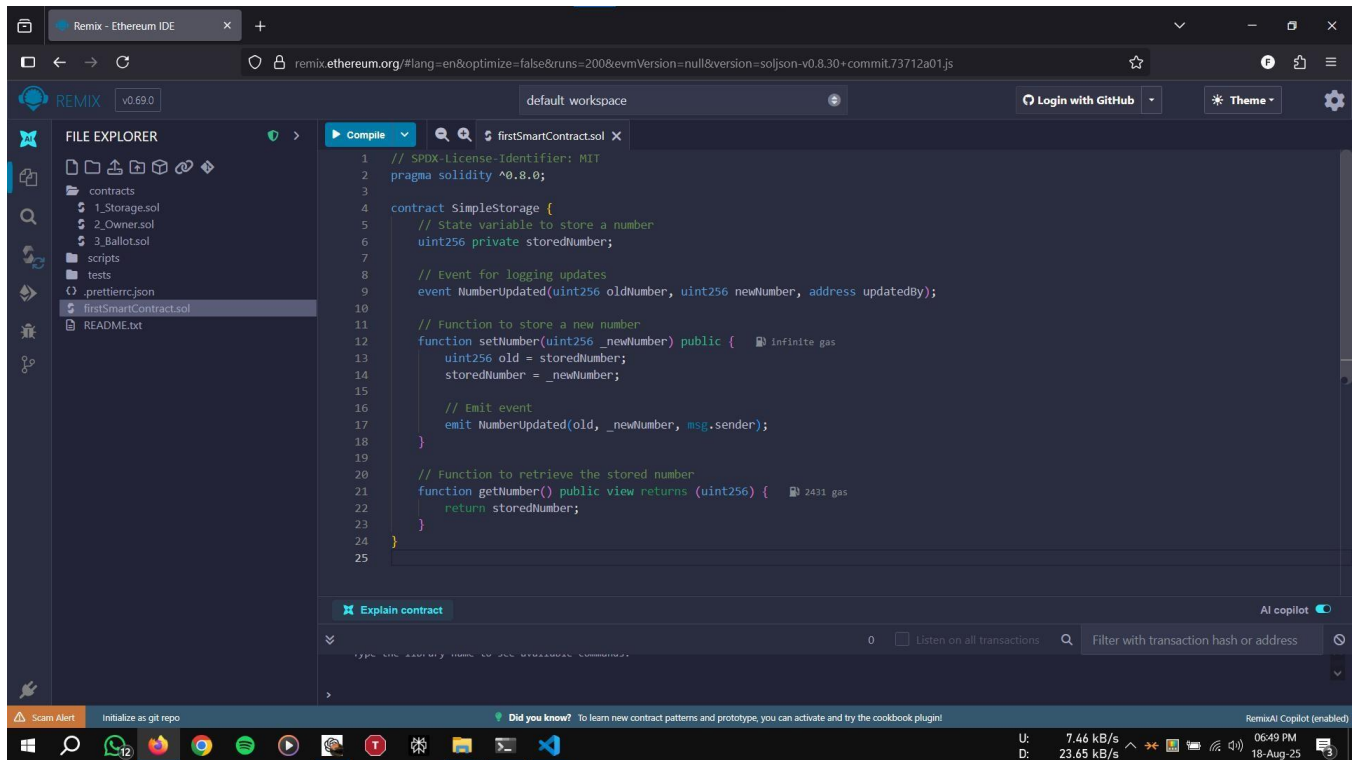
- Now open [remix.ethereum.org](https://remix.ethereum.org) where we can create and store our smart contract it is a ethereum IDE that we can write smart contract with help of solidity language.



### Step-02:

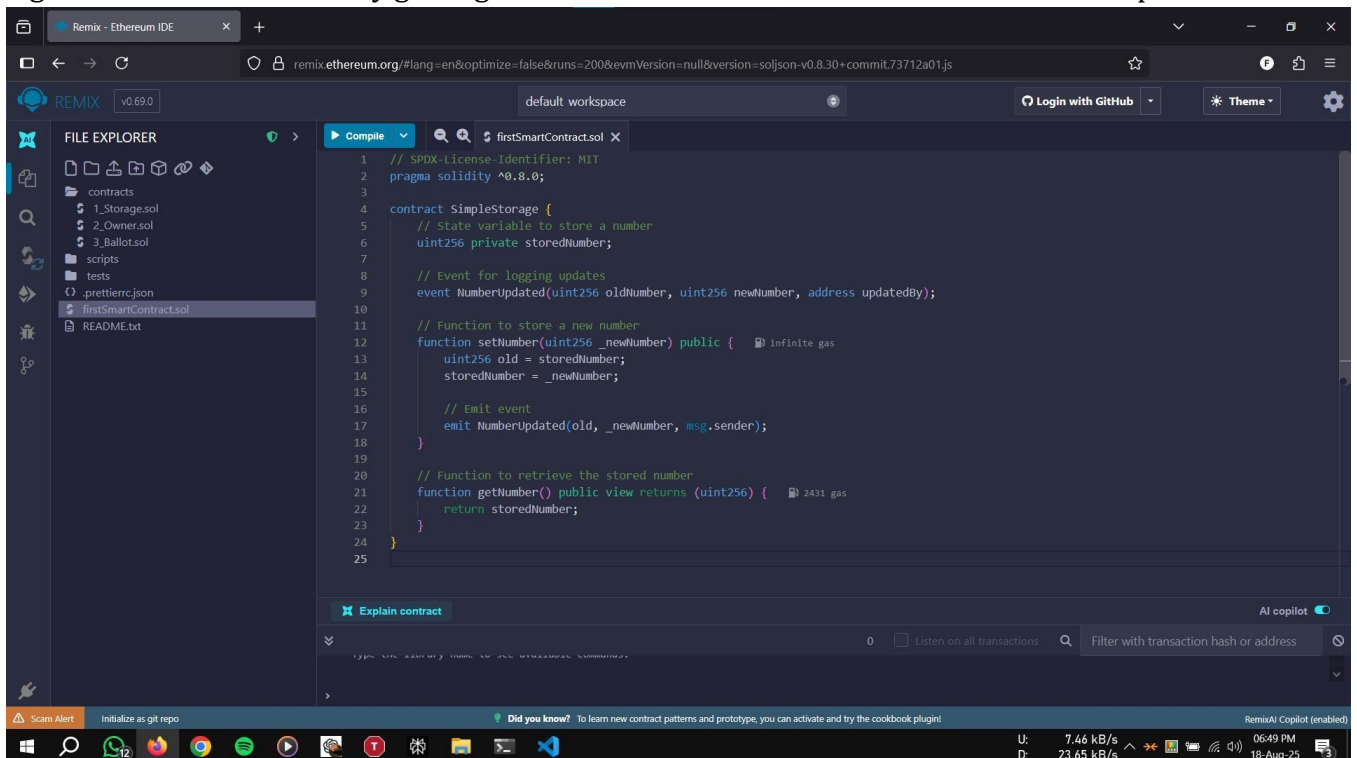
- Now create a new file from the top right corner while creating new file give the extension of the file **.sol** by giving **.sol** extension its helps remix to identify that the smart contract is written in solidity programming language.





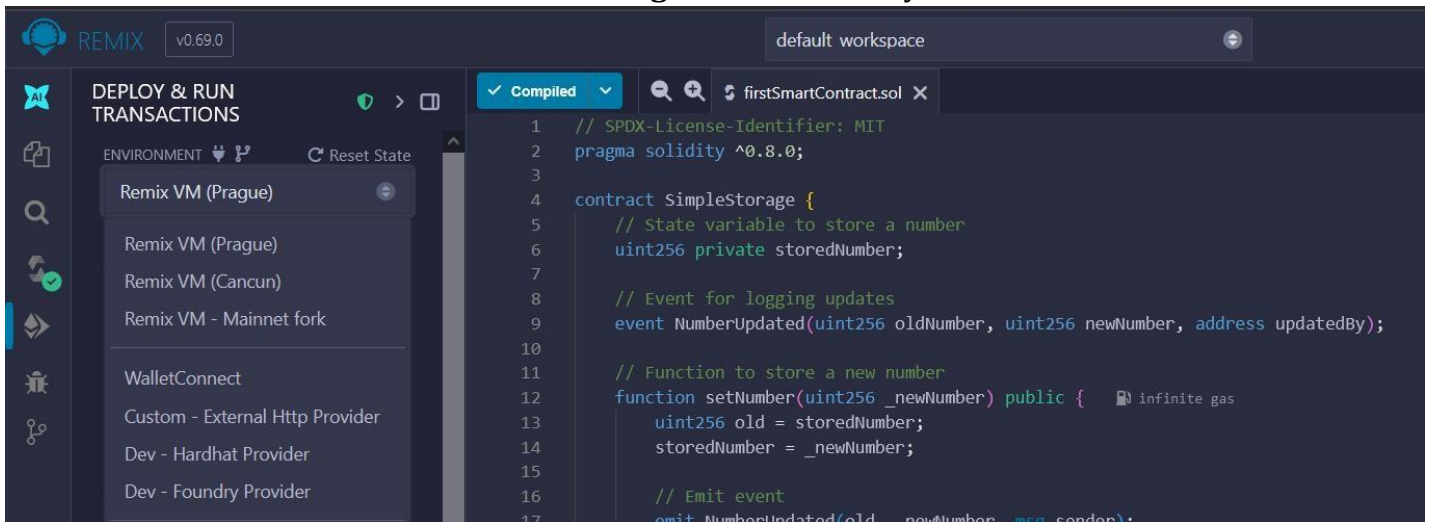
### Step-03:

- Now after writing the contract compile the contract after compiling the contract we should get a green color check mark by getting it we can know that the smart contract has compiled successfully.



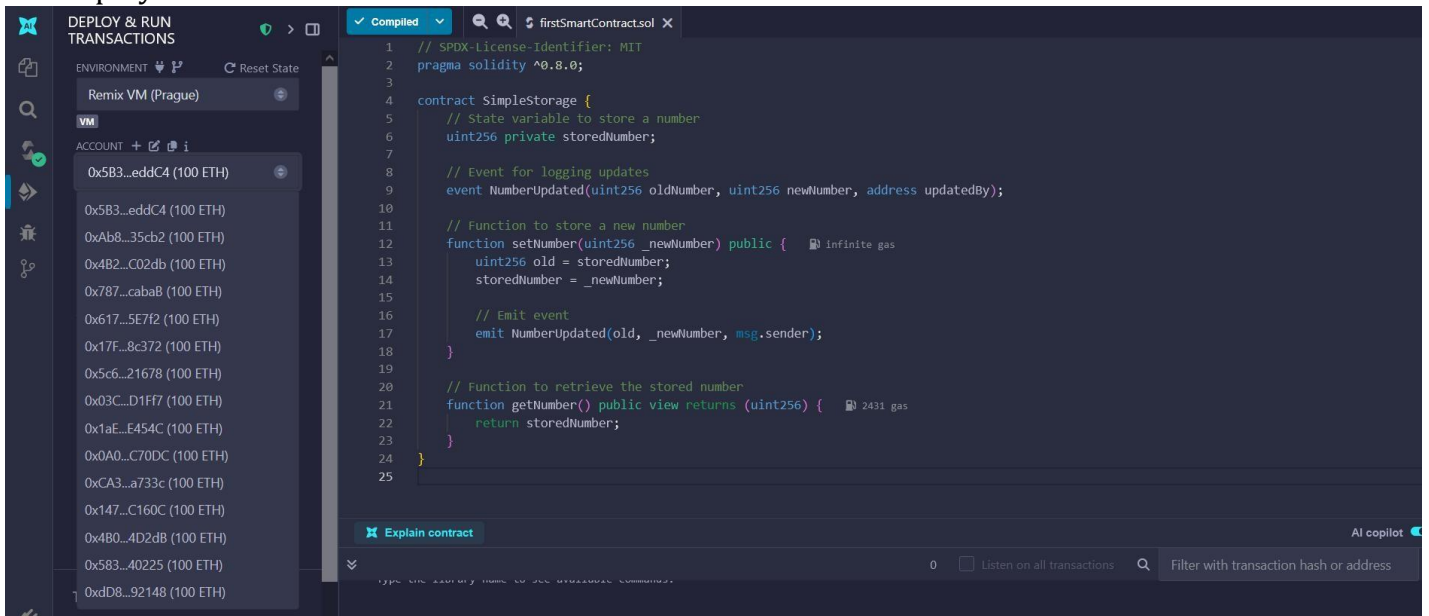
## Step -04:

- Now after successful compilation of the contract go to the section deployment on the left side taskbar.
- In that section select we have to set the environment to deploy the smart contract such as java VM but we are deploying this contract directly on the Ethereum testnet.
- So click on the environment and select “**Prague**” to connect to your wallet.



## Step-05:

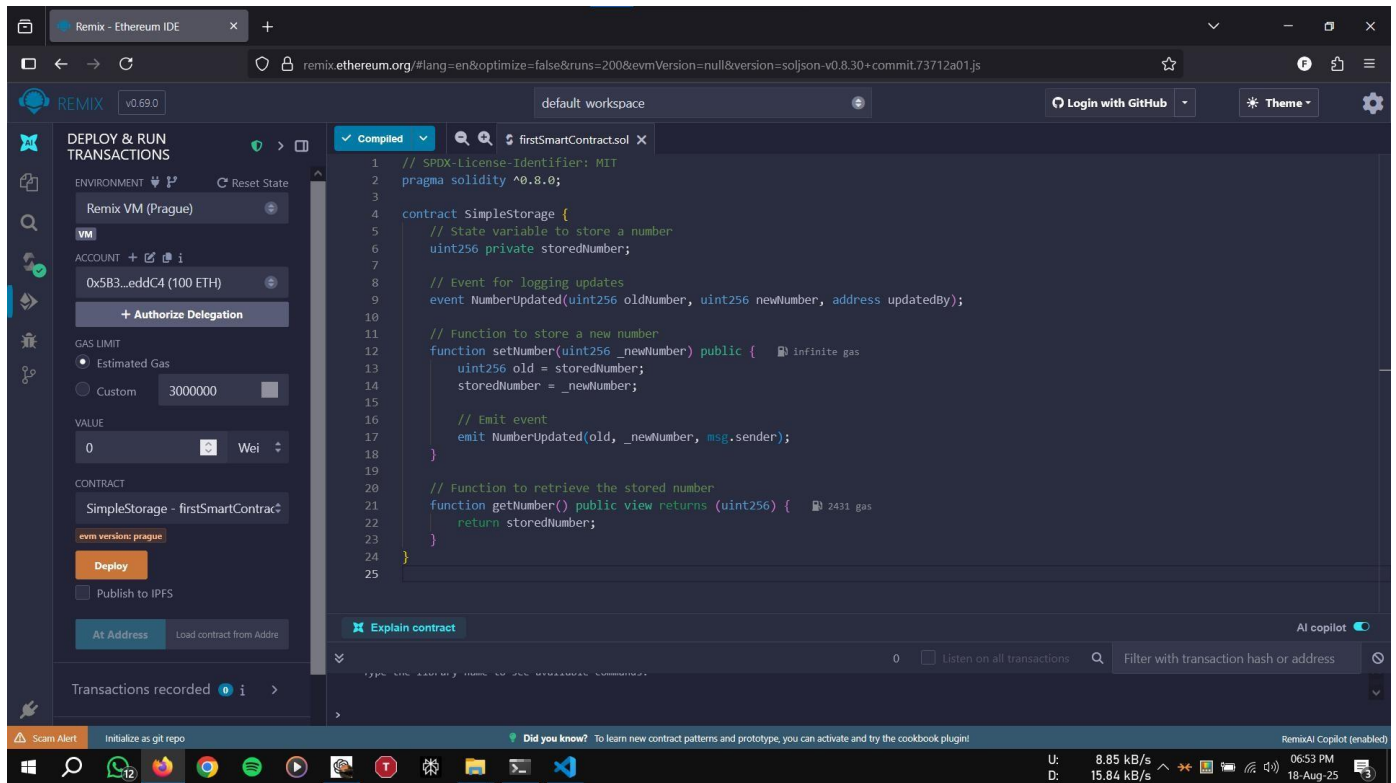
- After setting the environment we have select the account which acts as a sender of the deployment transaction.





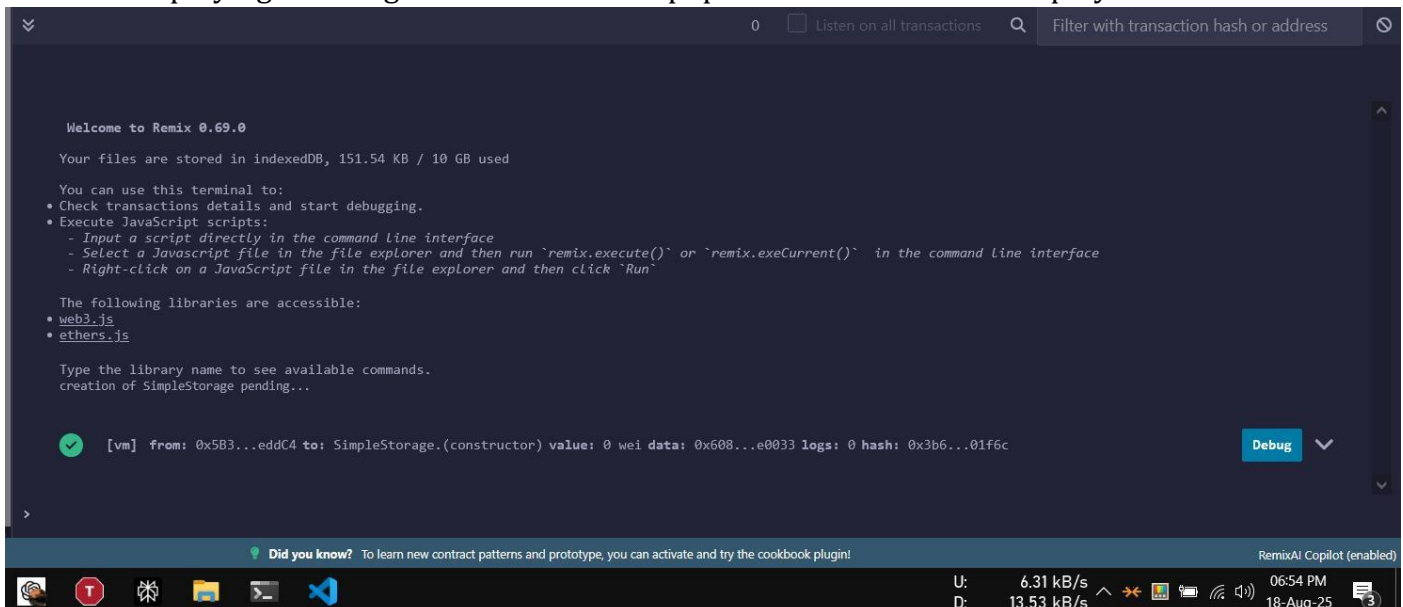
## Step-06:

- After setting the account also click on the deployment option by this we can successfully deploy our smart contract.



## Step-07:

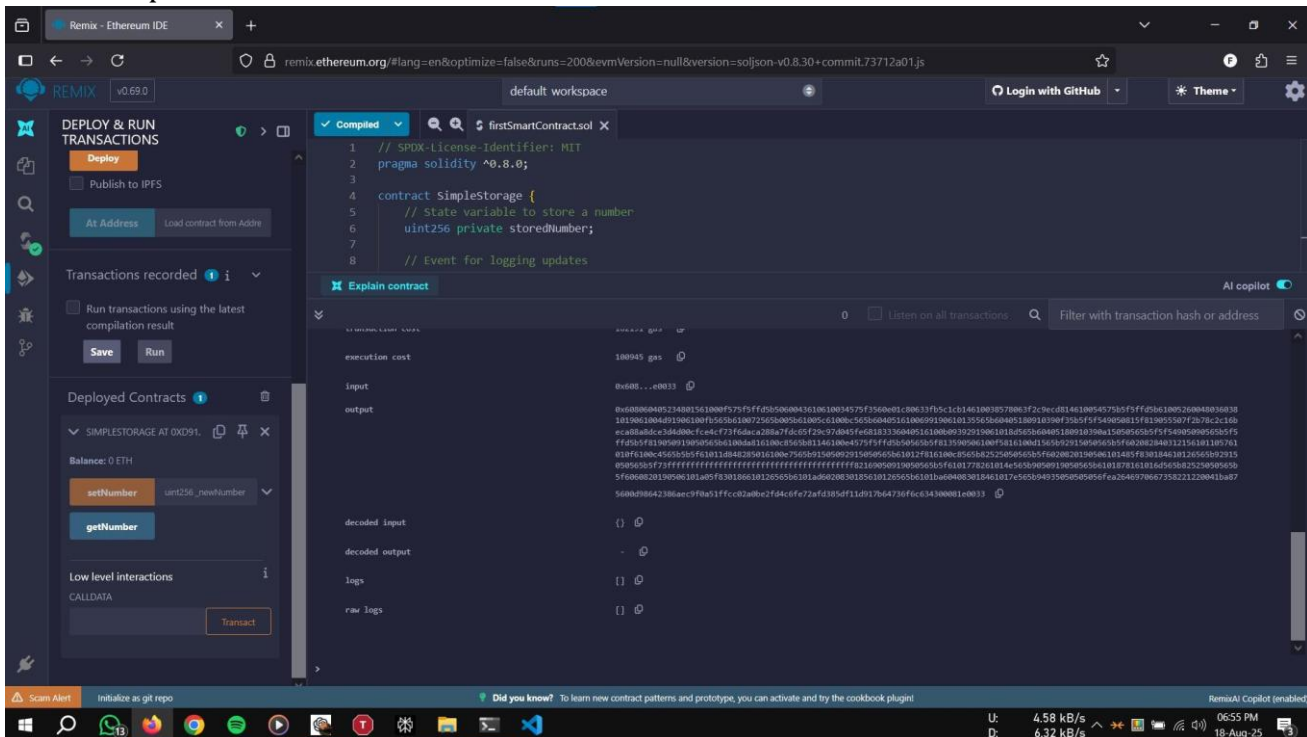
- After deploying we will get the confirmation pop that our contract has deployed in the terminal.



### 3.Interact with the Deployed

#### Contract Step-8:

- Once the contract developed you can interact with it with the help of its user interface that remix provides.



#### Conclusion:

By performing this practical we get to know what are smart contract and their working .we learnt to develop and deploy the our first smart contract using remix Ethereum IDE .we can interact with the smart contract for future of further purposes.

## Practical No - 3

### Aim: Identifying and Mitigating Common Vulnerabilities.

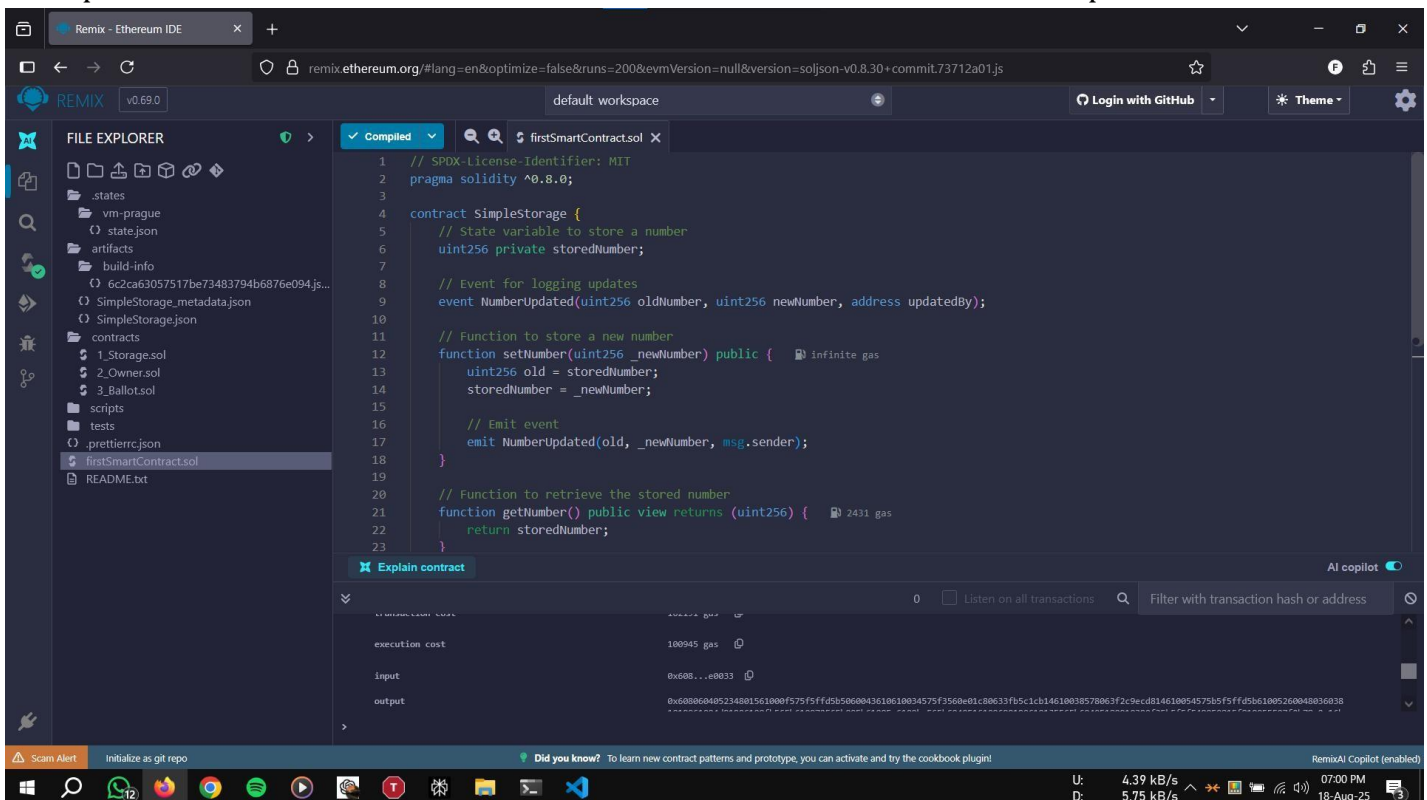
1. Exploring common security vulnerabilities in blockchain.
2. Implementing security best practices.
3. Conducting code reviews for security.

**SOL:** Identifying and mitigating common vulnerabilities in blockchain systems. This manual covers the exploration of common security vulnerabilities, the implementation of security best practices, and conducting code reviews for security.

### 1. Exploring Common Security Vulnerabilities in Blockchain.

#### Step 1:

- Open remix and create a new file with extension. sol as we have done in the practical 2.

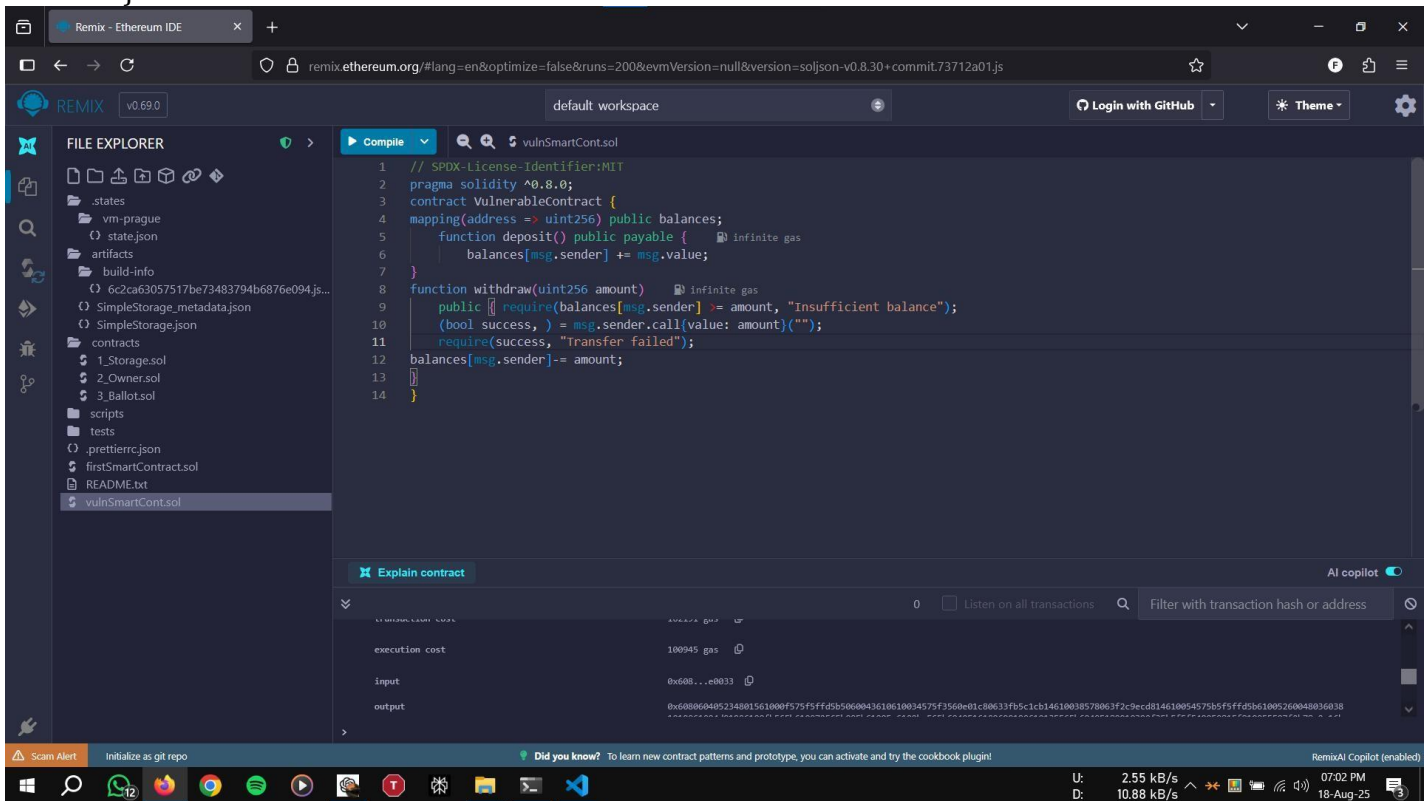


## Step 2:

- Write a Vulnerable Smart Contract
- Create a Smart Contract with Common Vulnerabilities:

```
pragma solidity ^0.8.0; contract
VulnerableContract {
    mapping(address => uint256) public balances; function deposit()
    public payable { balances[msg.sender] += msg.value;
}

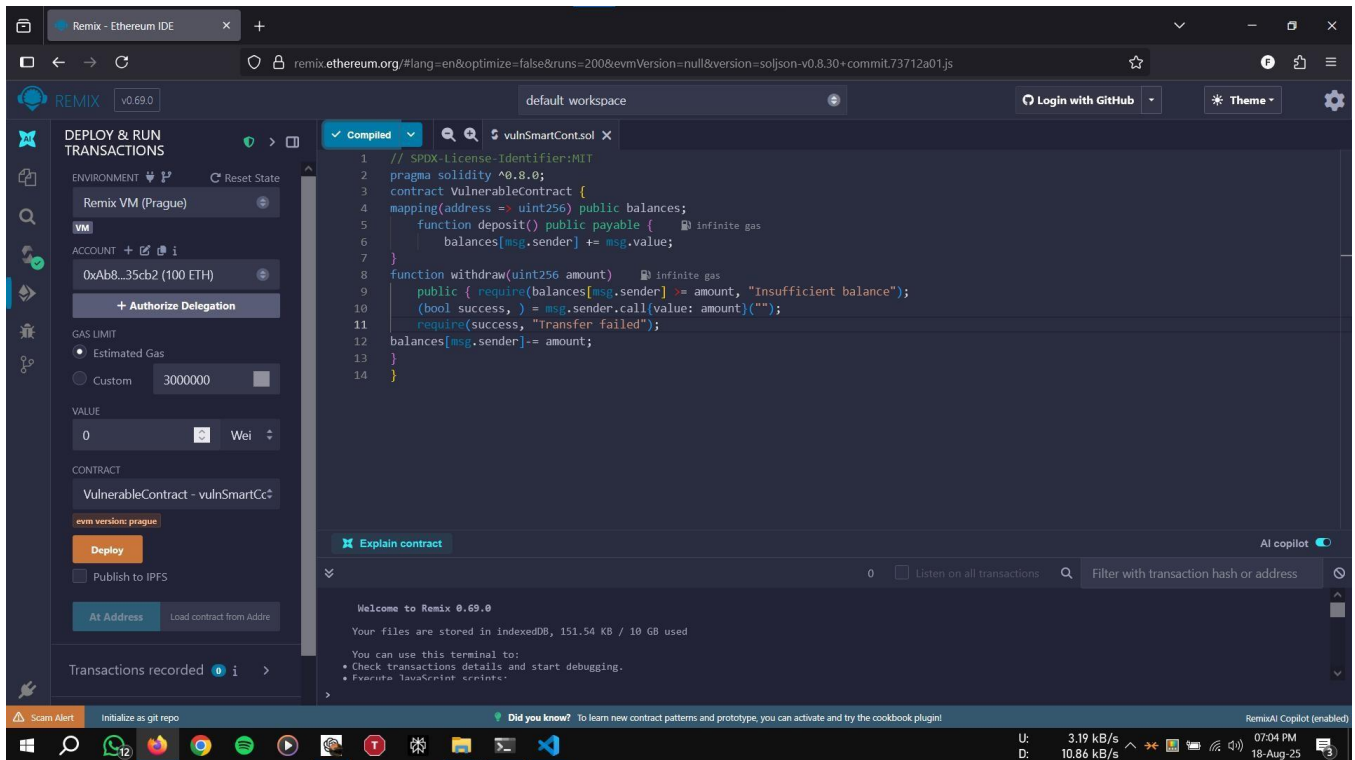
function withdraw(uint256 amount) public { require(balances[msg.sender] >= amount,
    "Insufficient balance"); (bool success, ) = msg.sender.call{value: amount}("");
    require(success, "Transfer failed");
    balances[msg.sender]-= amount;
}
}
```



## Step-03

### Deploy the Smart Contract:

- Compile the contract using the Remix IDE compiler.
- Deploy the contract using the Remix IDE.



## Step 4:

### Identify Vulnerabilities

#### 1. Analyze the Contract for Vulnerabilities:

- Identify the Reentrancy Vulnerability: In the `withdraw` function, the balance is updated after sending Ether, making it susceptible to reentrancy attacks.
- Identify the Integer Overflow/Underflow: Ensure all arithmetic operations are safe from overflow or underflow, especially in versions of Solidity before 0.8.0.

## 2. Implementing Security Best

### Practices Step 4:

#### Fix the Vulnerabilities

##### 1. Mitigate Reentrancy:

```
function withdraw(uint256 amount) public {
    require(balances[msg.sender] >= amount, "Insufficient balance");
    balances[msg.sender]-= amount;

    (bool success, ) = msg.sender.call{value: amount}(""); require(success,
    "Transfer failed");
}
```

## 2. Use SafeMath Library (for Solidity versions below 0.8.0):

```
//import "https://github.com/OpenZeppelin/openzeppelin  
contracts/blob/master/contracts/utils/math/SafeMath.sol";  
// using SafeMath for uint256;  
  
// function deposit() public payable {  
// balances[msg.sender] = balances[msg.sender].add(msg.value);  
// }
```

### Step 5:

#### Add Security Measures

##### 1. Implement Security Best Practices:

- Use Latest Solidity Version: Ensure you are using the latest stable version of Solidity.
- Check External Calls: Always check the return value of external calls.
- Limit Gas: Use `transfer` or `send` instead of `call` to limit gas and mitigate reentrancy.
- Use Modifiers: Implement access control using `onlyOwner` or similar modifiers.

## 3. Conducting Code Reviews for Security

### Step 6: Perform a Security Audit

##### 1. Review Code for Common Vulnerabilities:

- Ensure no presence of reentrancy, integer overflow/underflow, unchecked external calls, and other known issues.
- Follow a checklist to ensure all security best practices are implemented.

##### 2. Use Automated Tools:

- MythX: Use MythX for automated smart contract security analysis.
- Slither: Use Slither for static analysis and vulnerability detection.
- Oyente: Use Oyente for symbolic execution and security analysis.

### Step 7: Conduct Manual Code Review

##### 1. Peer Review:

- Have another developer review the code for potential vulnerabilities and best practices compliance.
- Discuss any findings and apply necessary fixes.

##### 2. Document Findings:

- Document all identified vulnerabilities and the steps taken to mitigate them.
- Ensure the documentation is thorough for future reference and audits

### Conclusion:

By following this lab manual, you will have explored common security vulnerabilities in blockchain, implemented security best practices, and conducted thorough code reviews to identify and mitigate potential security issues. This process helps ensure the robustness and security of your blockchain applications.