

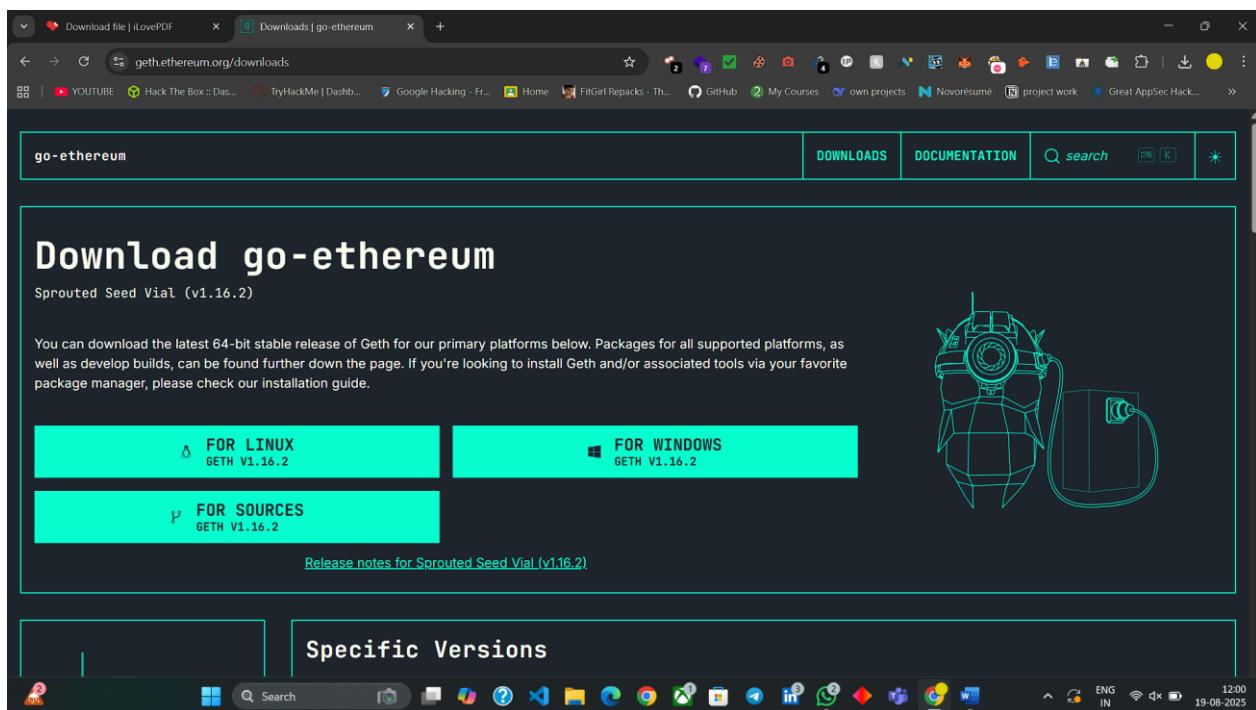
Practical No - 1

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.

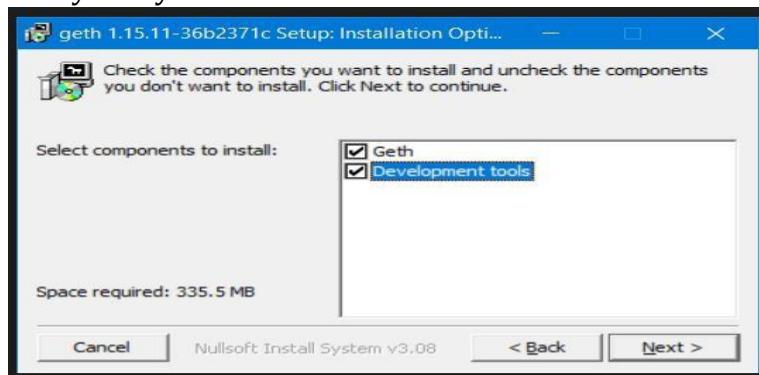
Step-01:

- 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 folder we will start to build our 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|12:03:10.764] 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
fffer=256.00MiB history="last 90000 blocks"
provided=1024 updated=4096
ETH=50 total=50
provided=4096 updated=2507
cap=50,000,000
backend=gokzg
clear=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\Ether
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
12:03
19-08-2025
```

Step-04:

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

```
{
  "config": {
    "chainId": 987,
    "homesteadBlock": 0,
    "epi150Block": 0,
    "epi155Block": 0,
    "epi158Block": 0,
    "byzantiumBlock": 0,
    "constantinopleBlock": 0,
    "petersburgBlock": 0,
    "istanbulBlock": 0,
    "terminalTotalDifficulty": 0
  },
  "difficulty": "0x400",
  "gasLimit": "0x8000000",
  "alloc": {}
}
```

Step-05:

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

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   "eip158Block": 0,
7   "byzantiumBlock": 0,
8   "constantinopleBlock": 0,
9   "petersburgBlock": 0,
10  "istanbulBlock": 0,
11  "terminalTotalDifficulty": 0
12 },
13 "difficulty": "0x400",
14 "gasLimit": "0x8000000",
15 "alloc": {}
16
17
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
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        database="Z:\Sem 7 Files\Class work\BCWS\ethereumPersonal\geth\chaindata" clean=16.00MiB dirty=0
INFO [08-18|18:38:07.404] Allocated cache and file handles                   database="Z:\Sem 7 Files\Class work\BCWS\ethereumPersonal\geth\chaindata" clean=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"
clientVersion=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": {}
17
18
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
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 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                database="Z:\Sem 7 Files\Class work\BCWS\ethereumPersonal\geth\chaindata" clean=154.00MiB dirty=256.00MiB
INFO [08-18|18:41:04.935] Allocated cache and file handles                  database="Z:\Sem 7 Files\Class work\BCWS\ethereumPersonal\geth\chaindata" clean=154.00MiB dirty=256.00MiB

```

- 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)

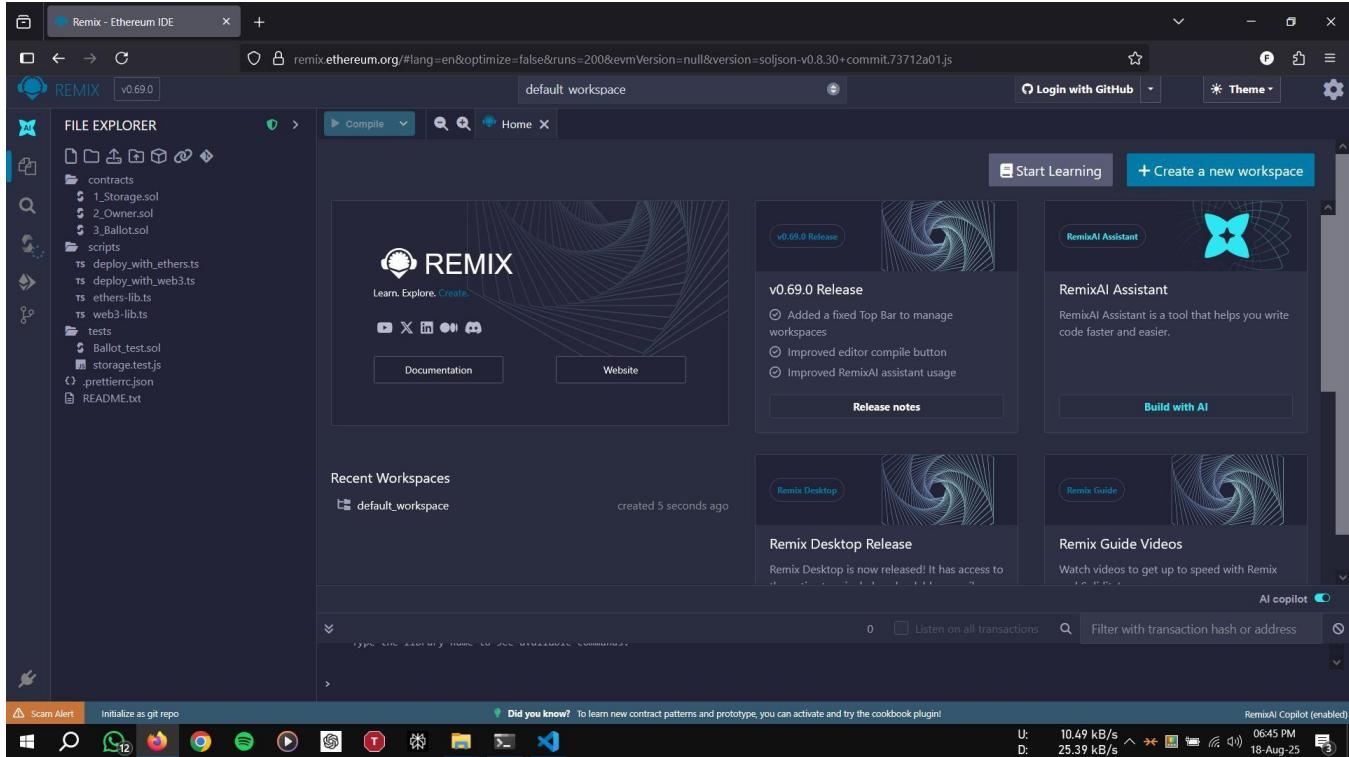
A Dapp is an application that operates in a decentralized manner. It utilizes smart contracts to store the application's business logic and state, replacing the traditional server-side component. However, it's important to note that DApps encompass more than just smart contracts. At the minimum, a DApp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

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.0;
3
4 contract SimpleStorage {
5     // State variable to store a number
6     uint256 private storedNumber;
7
8     // Event for logging updates
9     event NumberUpdated(uint256 oldNumber, uint256 newNumber, address updatedBy);
10
11    // Function to store a new number
12    function setNumber(uint256 _newNumber) public {
13        uint256 old = storedNumber;
14        storedNumber = _newNumber;
15
16        // Emit event
17        emit NumberUpdated(old, _newNumber, msg.sender);
18    }
19
20    // Function to retrieve the stored number
21    function getNumber() public view returns (uint256) {
22        return storedNumber;
23    }
24}

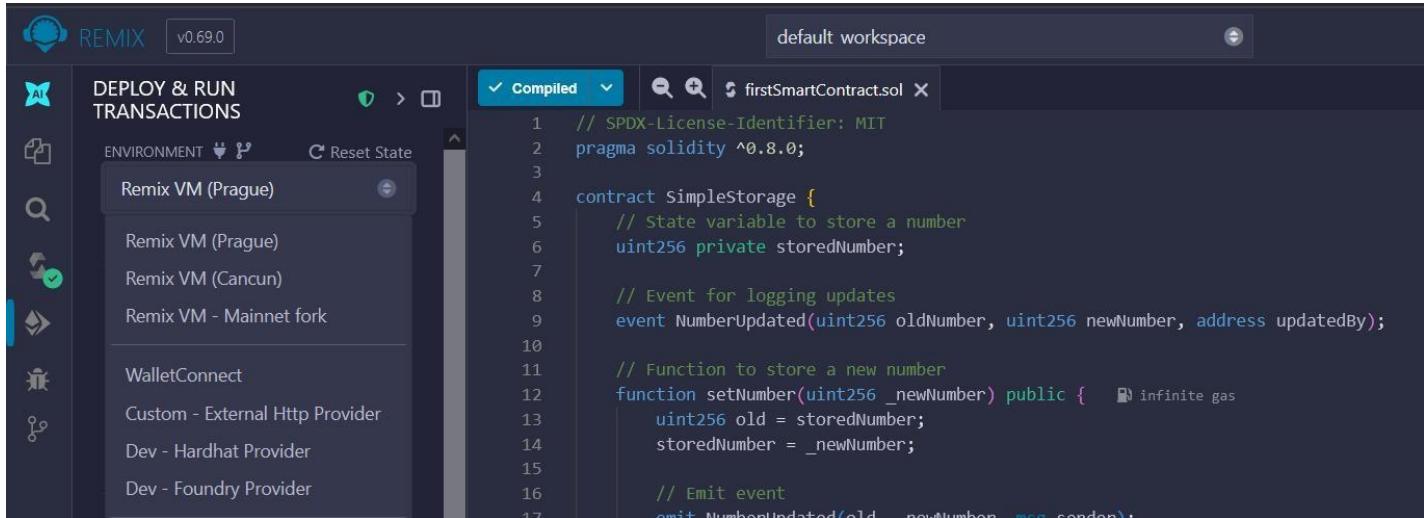
```

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 toolbar.
- 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.



The screenshot shows the Remix IDE interface. On the left, there's a sidebar with various icons and sections like 'DEPLOY & RUN TRANSACTIONS' and 'ENVIRONMENT'. Under 'ENVIRONMENT', 'Remix VM (Prague)' is selected. The main area is titled 'default workspace' and contains a code editor with Solidity code for a 'SimpleStorage' contract. The code includes functions for setting and getting a stored number, along with an event for logging updates.

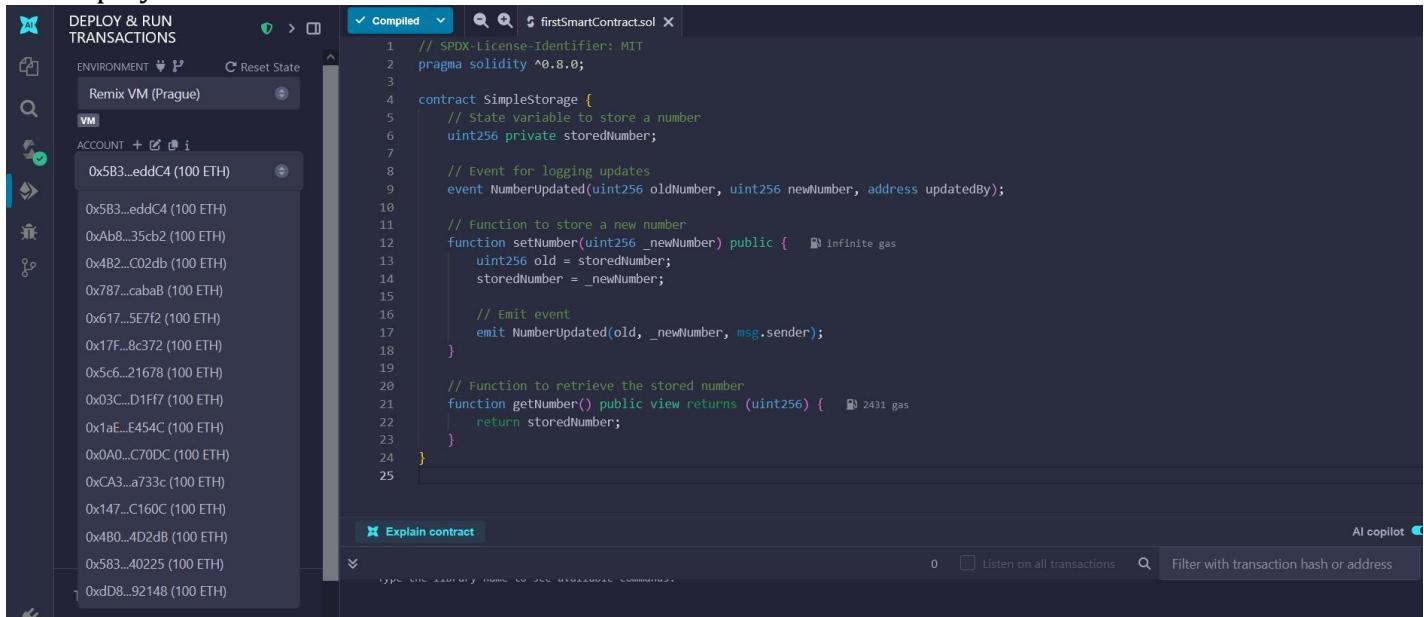
```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.0;
3
4 contract SimpleStorage {
5     // State variable to store a number
6     uint256 private storedNumber;
7
8     // Event for logging updates
9     event NumberUpdated(uint256 oldNumber, uint256 newNumber, address updatedBy);
10
11    // Function to store a new number
12    function setNumber(uint256 _newNumber) public {
13        uint256 old = storedNumber;
14        storedNumber = _newNumber;
15
16        // Emit event
17        emit NumberUpdated(old, _newNumber, msg.sender);
18    }
19
20    // Function to retrieve the stored number
21    function getNumber() public view returns (uint256) {
22        return storedNumber;
23    }
24}

```

Step-05:

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



The screenshot shows the Remix IDE interface with the 'ACCOUNT' section open in the sidebar. It lists several Ethereum accounts with their addresses and ETH balance. The account '0x5B3...eddC4 (100 ETH)' is currently selected. The main code editor area remains the same as in the previous screenshot, displaying the Solidity code for the 'SimpleStorage' contract.

Step-06:

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

The screenshot shows the Remix Ethereum IDE interface. On the left, the 'DEPLOY & RUN TRANSACTIONS' sidebar is open, showing the environment set to 'Remix VM (Prague)', an account selected (0x5B3...eddC4), and gas limit set to 3000000. The main workspace displays the Solidity code for 'firstSmartContract.sol'. The code defines a contract 'SimpleStorage' with a state variable '_storedNumber' and two functions: 'setNumber' and 'getNumber'. Both functions emit an event 'NumberUpdated'. The 'Deploy' button is highlighted. The bottom status bar shows network activity: U: 8.85 kB/s, D: 15.84 kB/s, and the date/time 18-Aug-25.

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

contract SimpleStorage {
    // State variable to store a number
    uint256 private storedNumber;

    // Event for logging updates
    event NumberUpdated(uint256 oldNumber, uint256 newNumber, address updatedBy);

    // Function to store a new number
    function setNumber(uint256 _newNumber) public {
        uint256 old = storedNumber;
        storedNumber = _newNumber;

        // Emit event
        emit NumberUpdated(old, _newNumber, msg.sender);
    }

    // Function to retrieve the stored number
    function getNumber() public view returns (uint256) {
        return storedNumber;
    }
}
```

Step-07:

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

The screenshot shows the Remix terminal window. It displays a welcome message for Remix 0.69.0 and information about indexedDB usage. It provides instructions for using the terminal, including running scripts and selecting files. The terminal shows a command being run: '[vm] from: 0x5B3...eddC4 to: SimpleStorage.(constructor) value: 0 wei data: 0x608...e0033 logs: 0 hash: 0x3b6...01f6c'. A 'Debug' button is visible at the top right. The bottom status bar shows network activity: U: 6.31 kB/s, D: 13.53 kB/s, and the date/time 18-Aug-25.

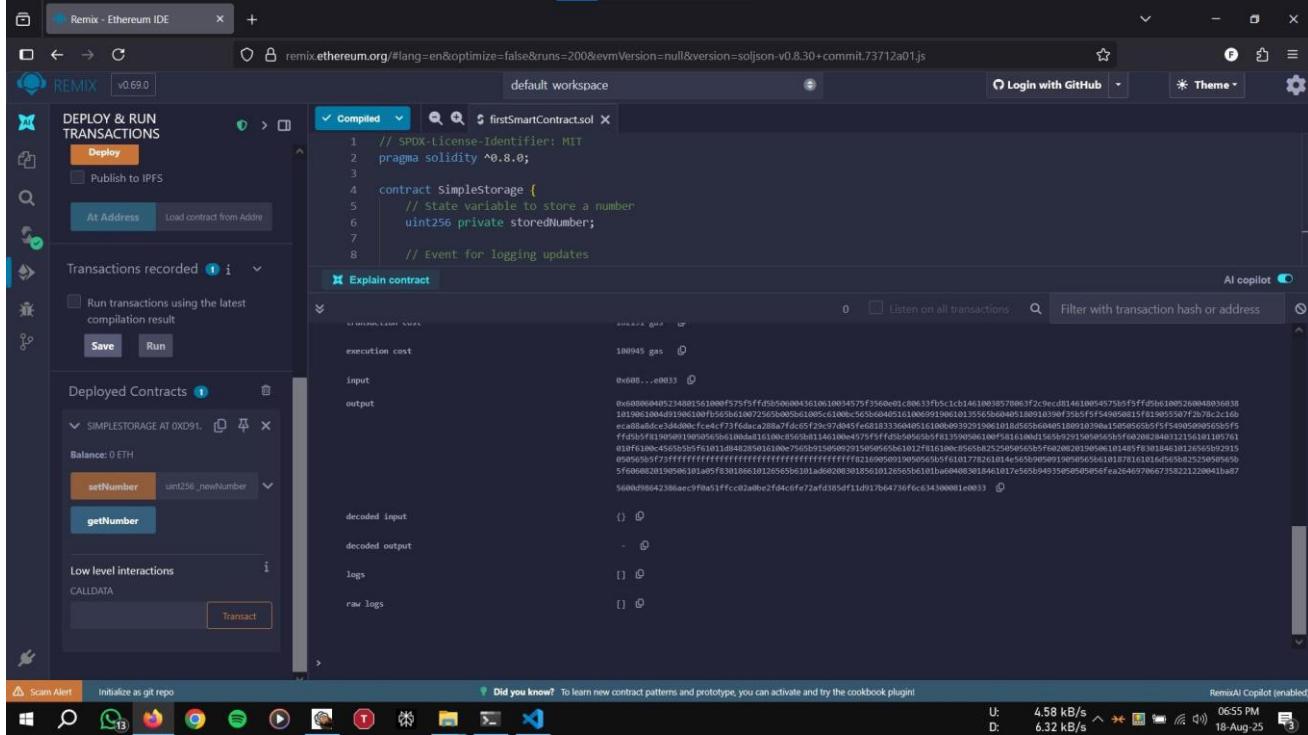
Welcome to Remix 0.69.0
Your files are stored in indexedDB, 151.54 KB / 10 GB used
You can use this terminal to:
• Check transactions details and start debugging.
• Execute JavaScript scripts:
- Input a script directly in the command Line interface
- Select a Javascript file in the file explorer and then run `remix.execute()` or `remix.exeCurrent()` in the command line interface
- Right-click on a JavaScript file in the file explorer and then click 'Run'
The following libraries are accessible:
• [web3.js](#)
• [ethers.js](#)
Type the library name to see available commands.
creation of SimpleStorage pending...

[vm] from: 0x5B3...eddC4 to: SimpleStorage.(constructor) value: 0 wei data: 0x608...e0033 logs: 0 hash: 0x3b6...01f6c

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.



The screenshot shows the Remix Ethereum IDE interface. On the left, the sidebar displays 'DEPLOY & RUN TRANSACTIONS' with a 'Deploy' button and 'At Address' dropdown. Below it is a 'Transactions recorded' section with a 'Run' button. Under 'Deployed Contracts', there is a list for 'SIMPLESTORAGE' with address 0x0D91, balance 0 ETH, and two buttons: 'setNumber' and 'getNumber'. A 'Low level interactions' section with a 'CALLDATA' tab and a 'Transact' button is also present. The main workspace shows the Solidity code for 'firstSmartContract.sol' and an 'Explain contract' panel. The code defines a simple storage contract with a state variable 'storedNumber' and an event for logging updates. The 'Explain contract' panel includes sections for 'execution cost' (108945 gas), 'input' (0x60...e0833), 'output' (a long hex string), 'decoded input' (empty), 'decoded output' (empty), 'logs' (empty), and 'raw logs' (empty). The bottom status bar shows network activity and the date/time.

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

contract SimpleStorage {
    // State variable to store a number
    uint256 private storedNumber;

    // Event for logging updates
    event Log(uint256 indexed _id, uint256 _value);
}
```

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.

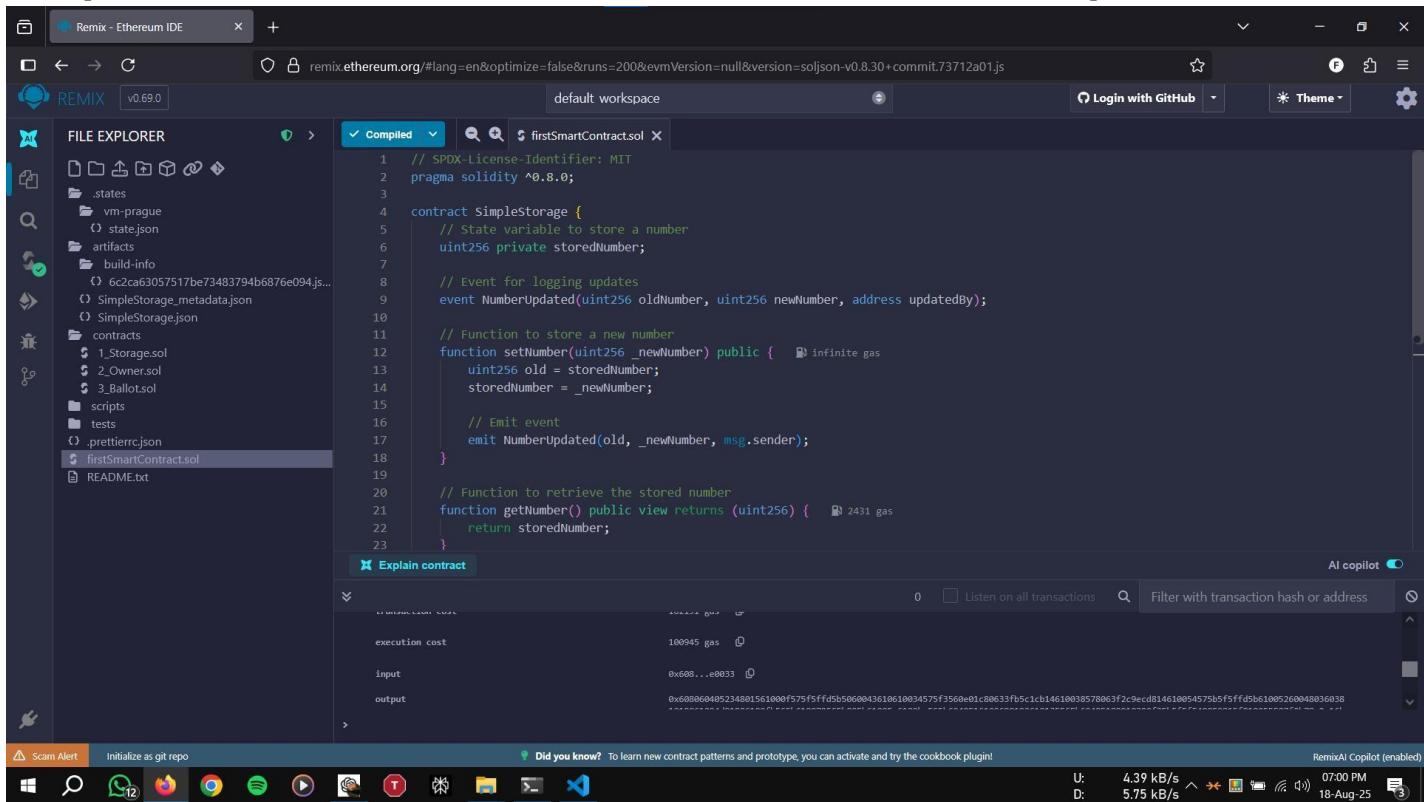
- 1.Exploring common security vulnerabilities in blockchain.
- 2.Implementing security best practices.
- 3.Conducting code reviews for security.

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.



The screenshot shows the Remix Ethereum IDE interface. The left sidebar is the 'FILE EXPLORER' showing files like .states, vm-prague, artifacts, contracts (Storage.sol, Owner.sol, Ballot.sol), scripts, tests, and .prettierrc.json. The main area is titled 'default workspace' and shows the 'Compiled' tab for 'firstSmartContract.sol'. The code is as follows:

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.0;
3
4 contract SimpleStorage {
5     // State variable to store a number
6     uint256 private storedNumber;
7
8     // Event for logging updates
9     event NumberUpdated(uint256 oldNumber, uint256 newNumber, address updatedBy);
10
11    // Function to store a new number
12    function setNumber(uint256 _newNumber) public {
13        uint256 old = storedNumber;
14        storedNumber = _newNumber;
15
16        // Emit event
17        emit NumberUpdated(old, _newNumber, msg.sender);
18    }
19
20    // Function to retrieve the stored number
21    function getNumber() public view returns (uint256) {
22        return storedNumber;
23    }

```

Below the code, there's an 'Explain contract' section with tabs for 'GENERAL CASES' and 'AVOIDABLE GAS'. It shows execution cost (100945 gas), input (0x60...e033), and output (0x608060405234801561000f57f5ff5b50600043610610034575f3560e01c80633fb5c1cb14610038578063f2c9ecd814610054575b5f5ffd5b1005260048036038). At the bottom, there are status icons for network connection, battery, and signal, along with system information: U: 4.39 kB/s, D: 5.75 kB/s, 07:00 PM, 18-Aug-25.

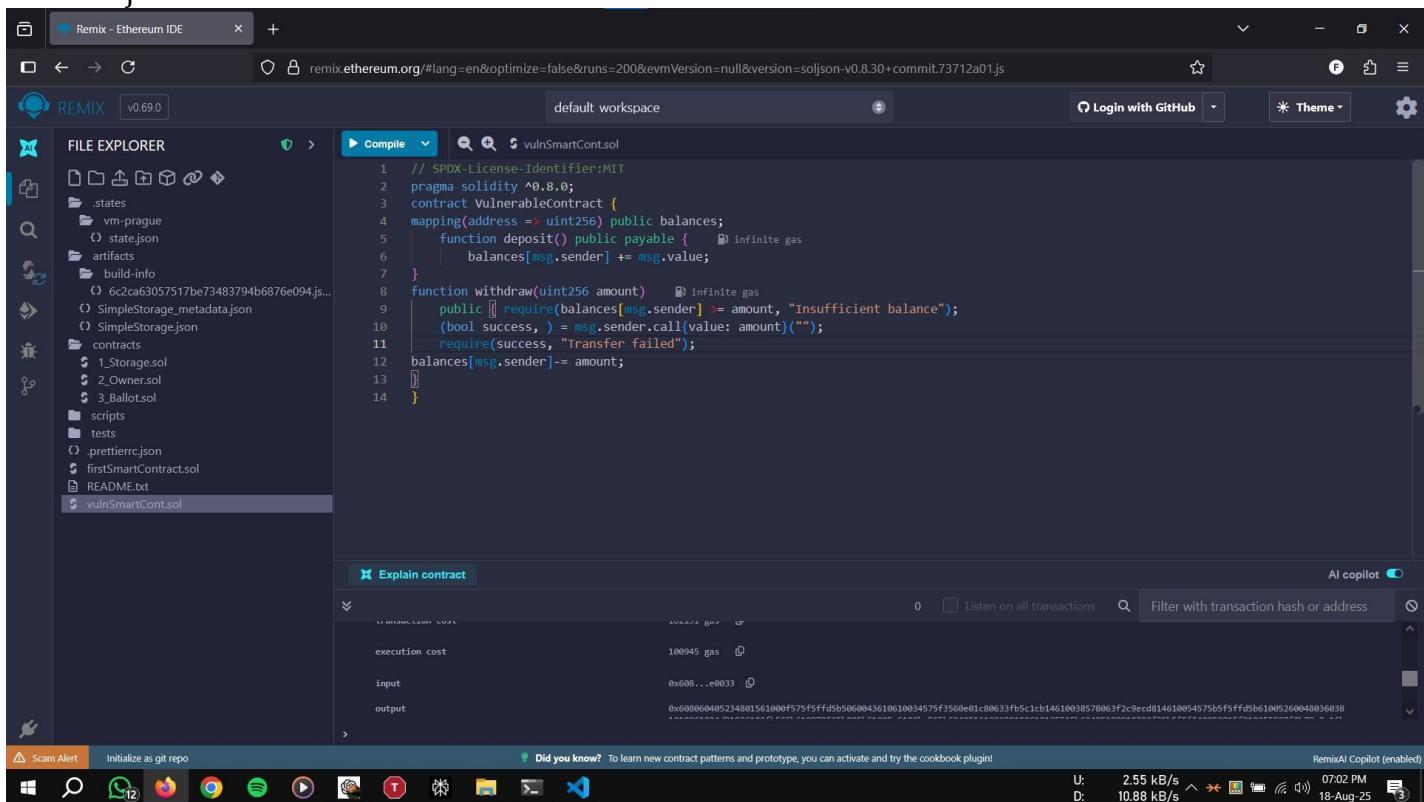
Step 2:

- 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.

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.8.0;
3 contract VulnerableContract {
4     mapping(address => uint256) public balances;
5     function deposit() public payable {
6         balances[msg.sender] += msg.value;
7     }
8     function withdraw(uint256 amount) public {
9         require(balances[msg.sender] >= amount, "Insufficient balance");
10        (bool success,) = msg.sender.call{value: amount}("");
11        require(success, "Transfer failed");
12        balances[msg.sender]-= amount;
13    }
14 }

```

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.