Task 3 BockChain

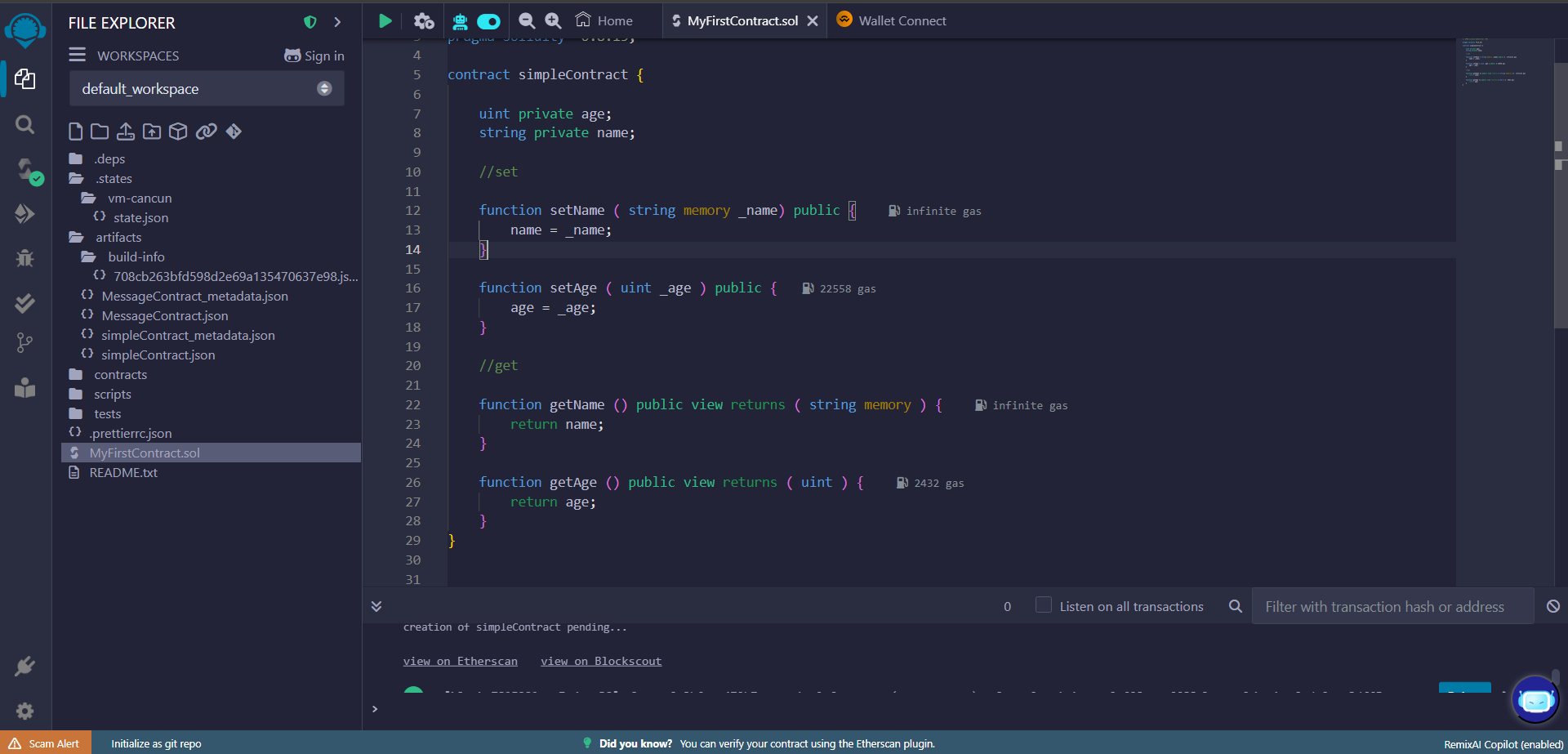
Assignment 1

1. Used Remix IDE.
2. It is browser based smart contract framework.
3. No installation required.
4. Created a .sol file named “MyFirstContact”.

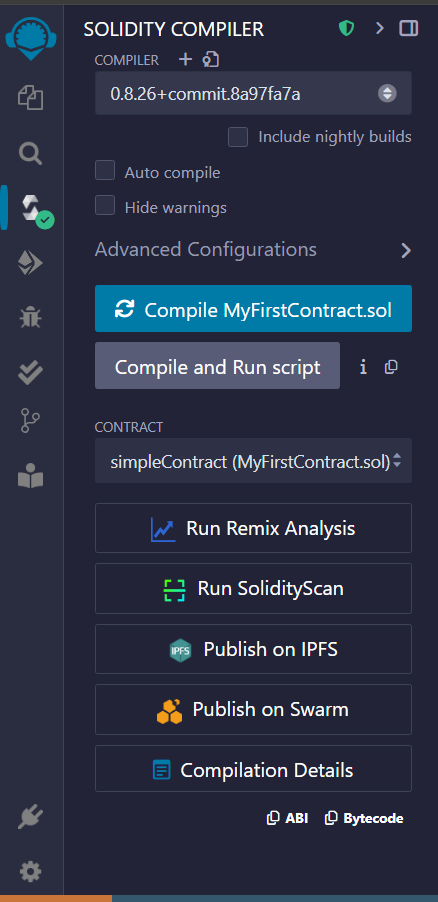
This Solidity code defines a simple smart contract named simpleContract. It has two private variables: age (an unsigned integer) and name (a string). The contract includes four functions:

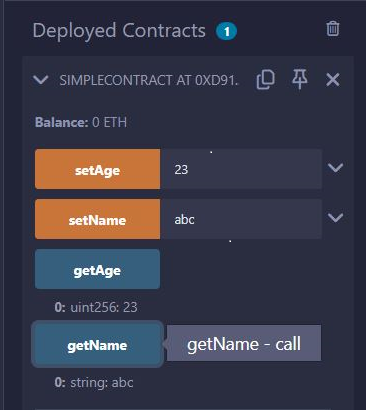
* **Setter Functions**:
  + setName: Sets the name variable.
  + setAge: Sets the age variable.
* **Getter Functions**:
  + getName: Returns the current name.
  + getAge: Returns the current age.

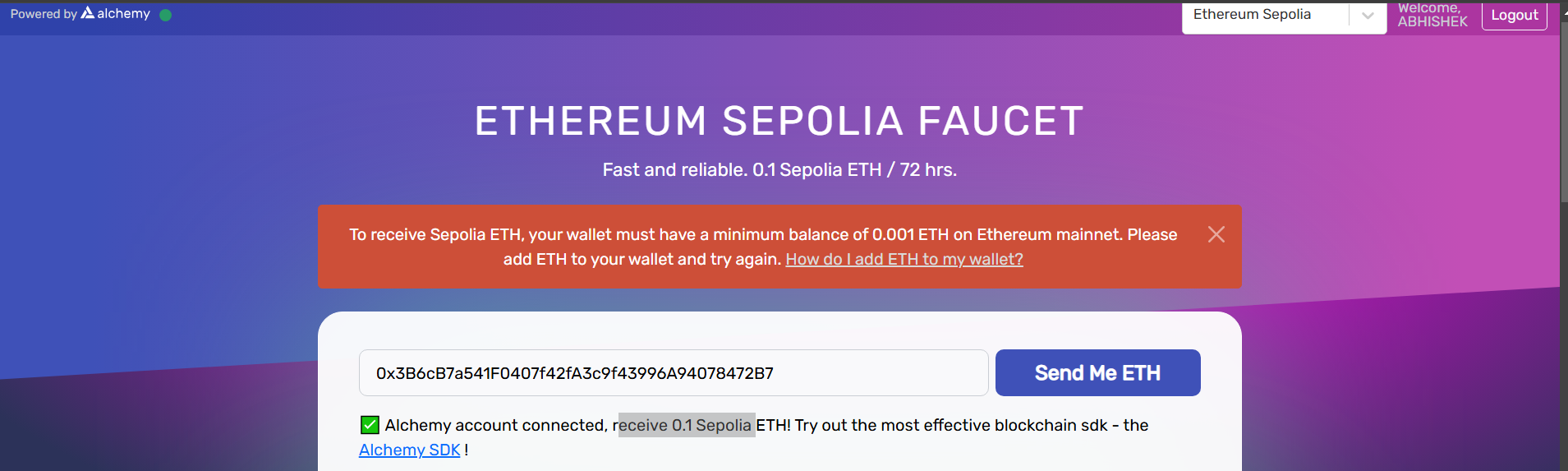
These functions allow users to modify and retrieve the contract's state. The view keyword in getter functions indicates that they do not modify the state of the contract.

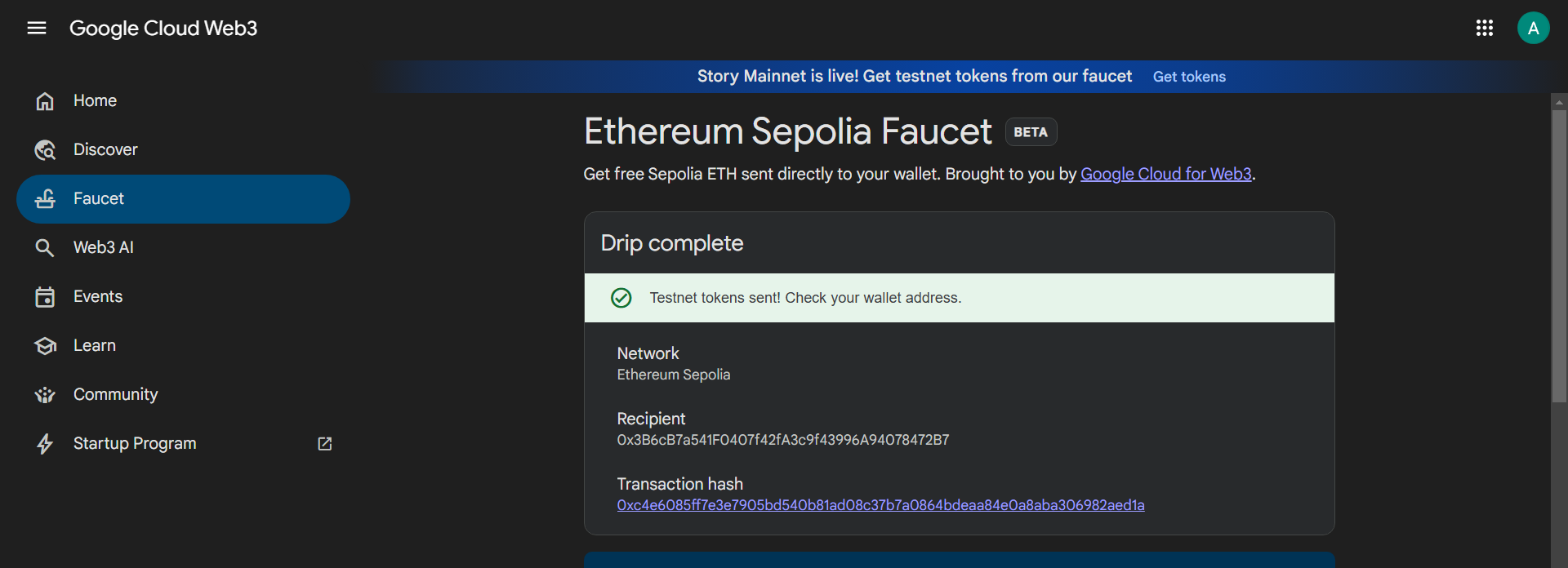


1. Manual compilation done (Auto Compile was not enabled)



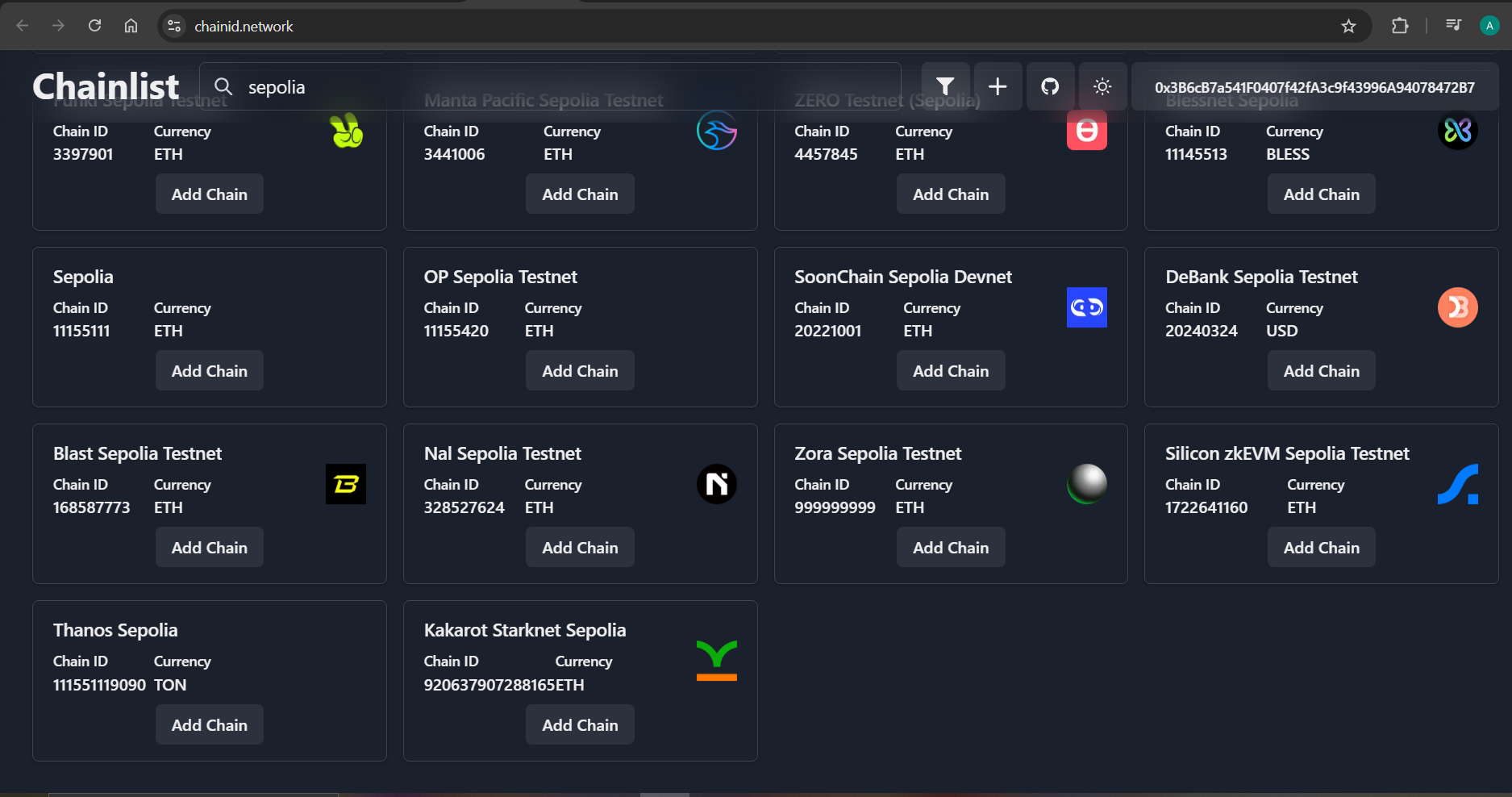
1. Deployed the contract using the Remix VM (Cancun) as the Environment. Reference : <https://youtu.be/kfKpXHy3pPg?si=FuvVqE4MQwuczsd->
2. After successful deployment, tried the setAge ,setName, getAge and getName functions.
3. Installed the MetaMask extension from ChromeStore.
4. Visited for faucet : <https://cloud.google.com/application/web3/faucet/ethereum/sepolia> to get SepoliaETH tokens (got help from a friend). Tried to make an Alchemy account and get the tokens, but failed since there is a minimum cap on ETH tokens one must initially have to avoid spammers.



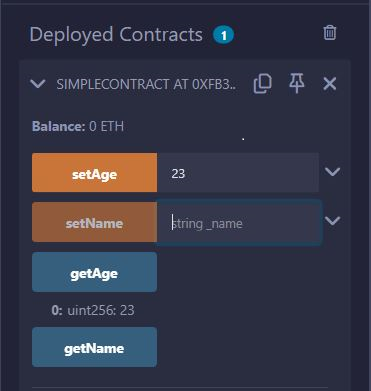
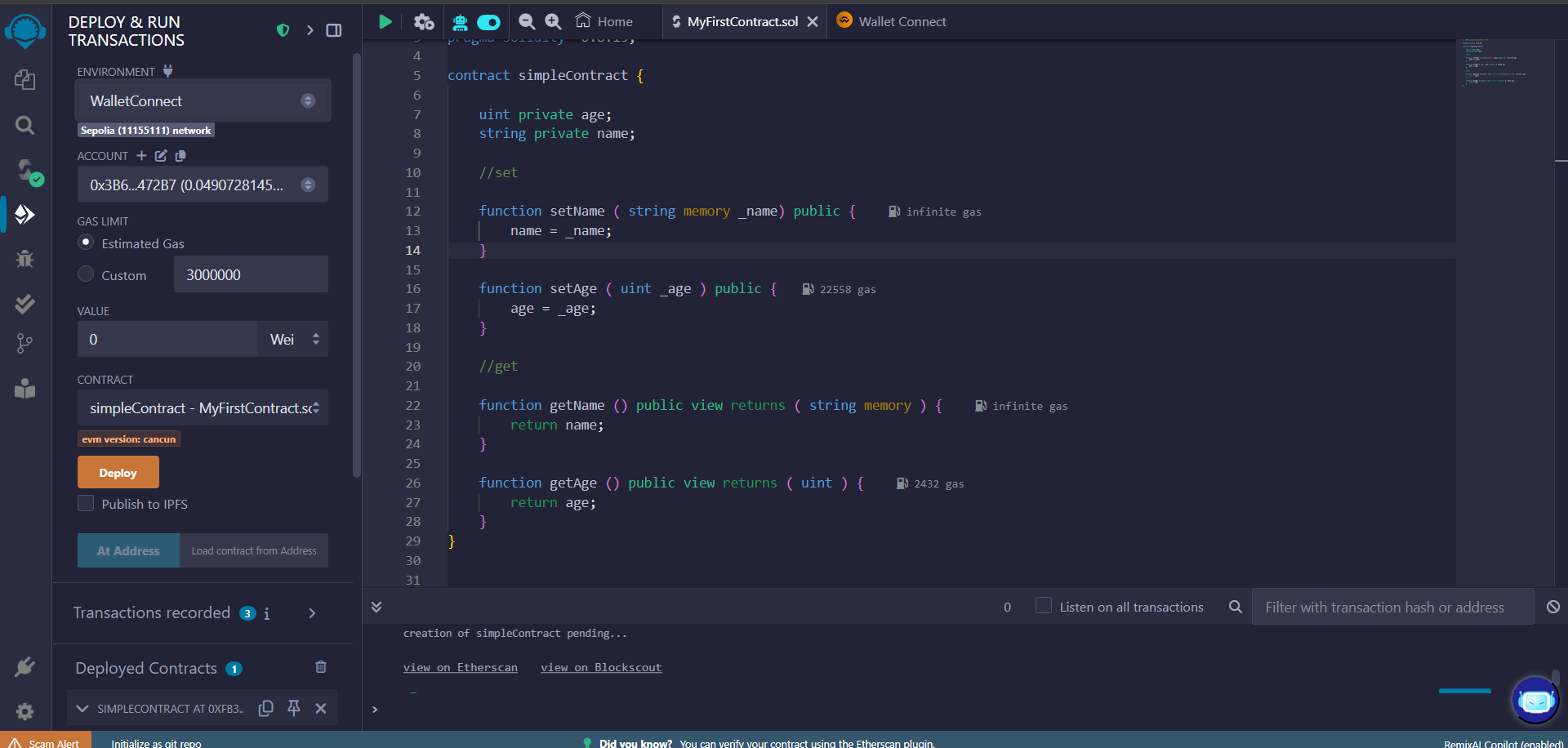


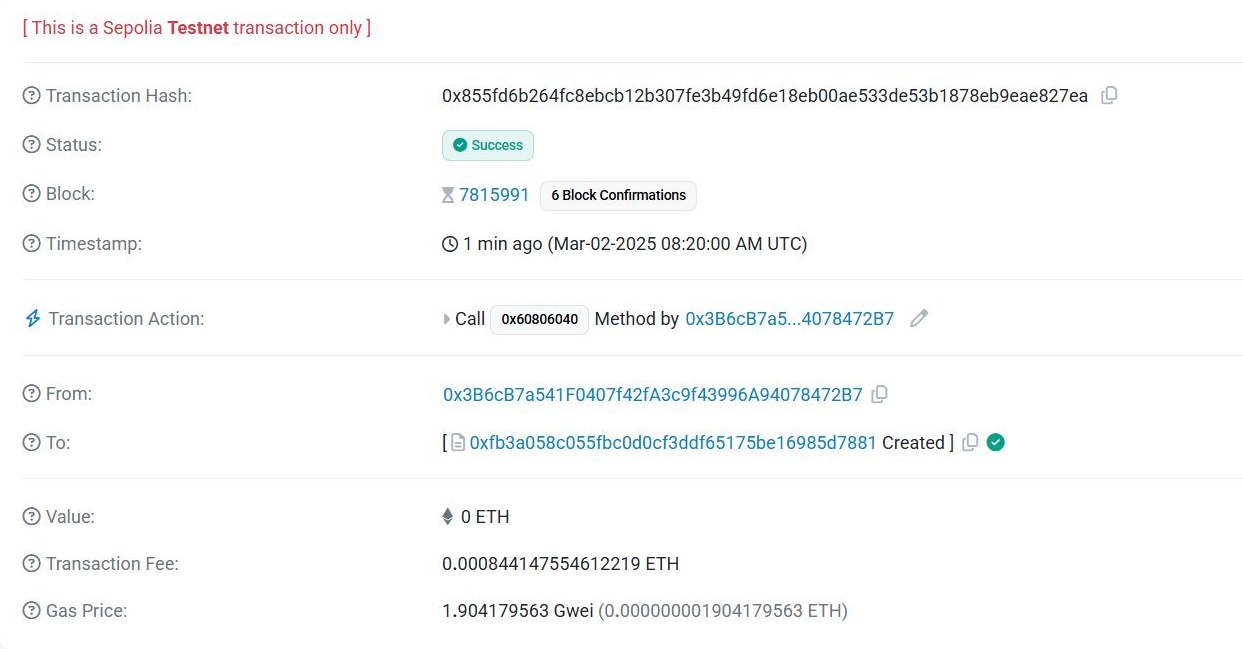
1. Now deployed the smart contact on sepolia testnet by first going to <https://chainid.network/> and searching “sepolia”.

Cl



Click on “Add Chain” and you get a prompt from MetaMask ,accept it.

1. Go to Remix Ide, select the environment WalletConnect,choose MetaMask. The video reference : <https://youtu.be/I_OZd0HN7ro?si=eaeXK54MBBjdHWox>. It mentions to select “Injected Provider -MetaMask”, but it was not there.
2. Some ETH around 0.0005 was charged(deducted) for deployment and for setAge it was around 0.0001. It took longer for deployment than previous case. 
3. Clicked on “View on Etherscan”



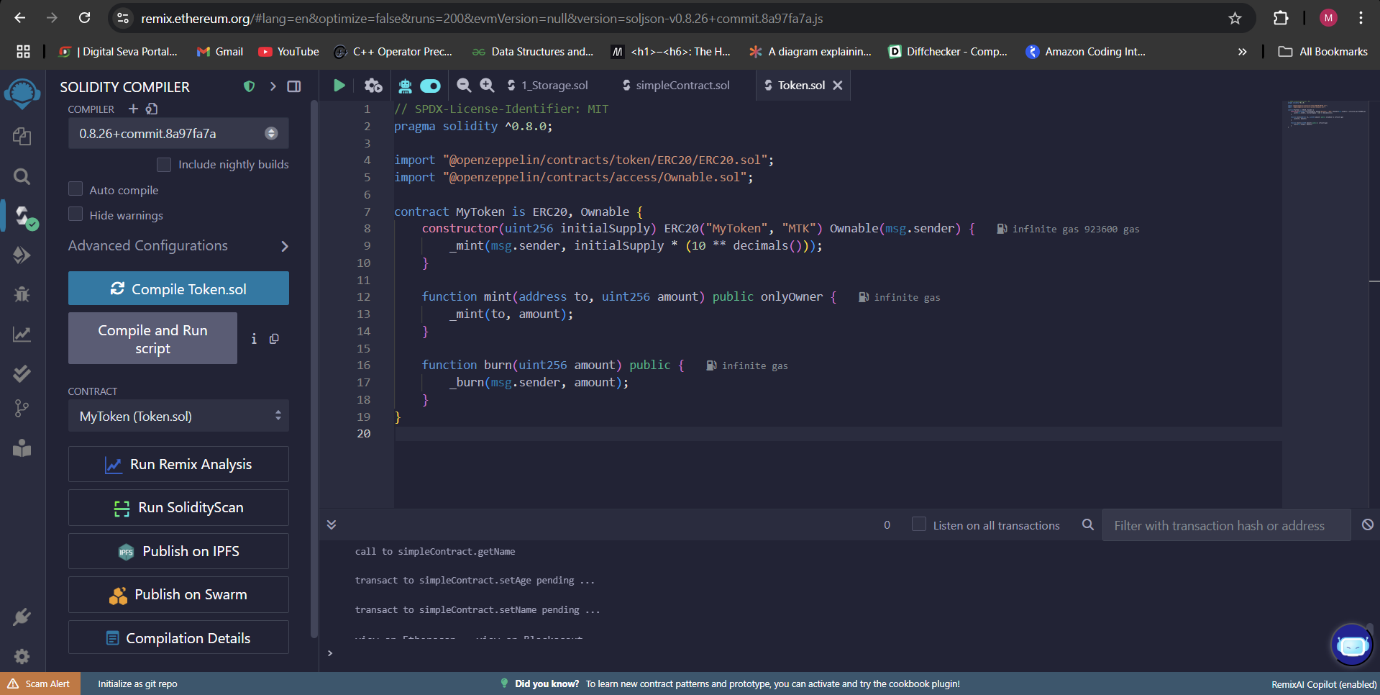
Obtained information about the transaction.

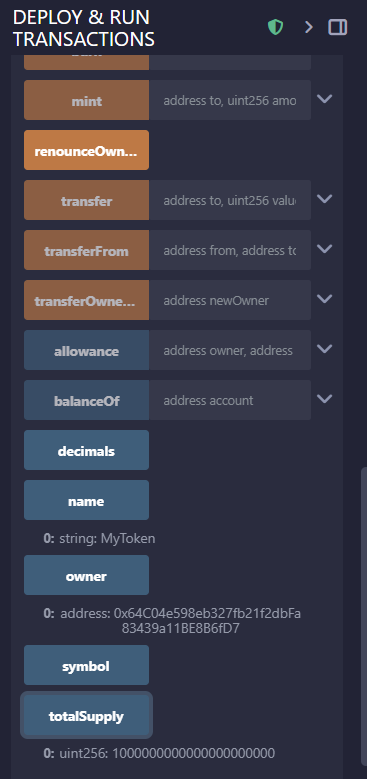
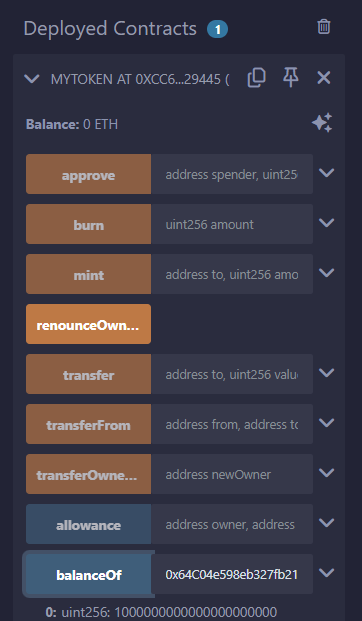
Github : <https://github.com/Ccode104/Blockchain>

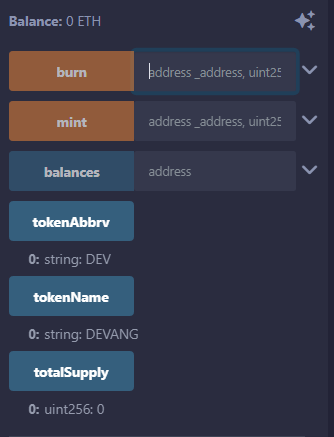
Assignment 2

1. Minting means to add tokens, Burning means to remove tokens, while transfer just changes the ownership of the tokens.
2. We can create our own token with a custom name and abbreviation as well.
3. 3 functions : mint,burn and transfer is defined. Total\_Supply variable tracks the total number of tokens.

Check the token name,abbreviation and total supply.

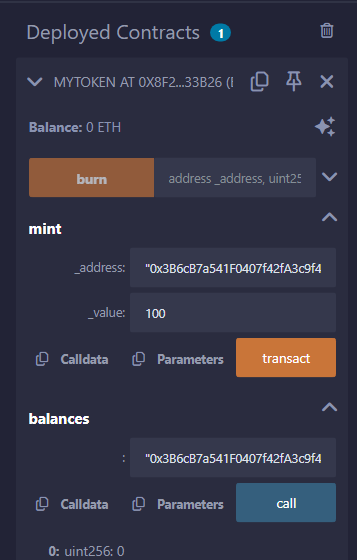




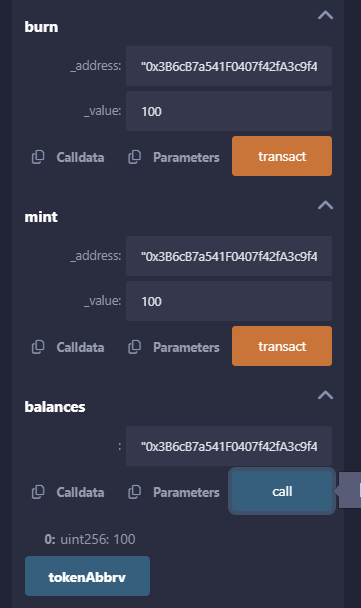


* Use balanceOf(your\_address) to check your balance.
* Use transfer(address, amount) to send tokens.
* Use mint(address, amount) to create more tokens.

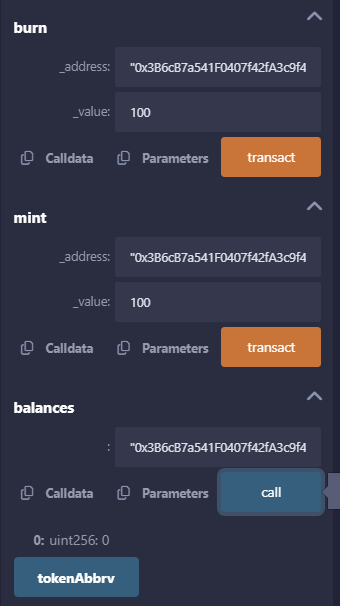
Mint 100 tokens to your address(or any other address)



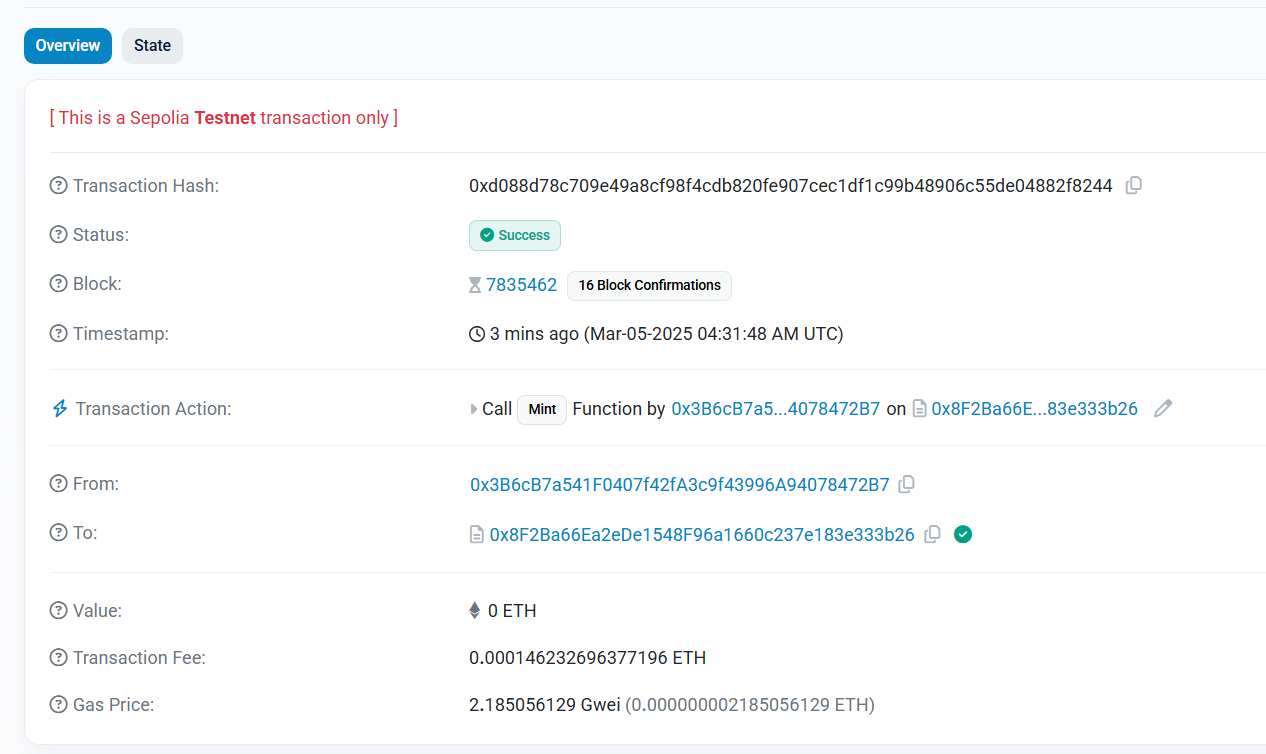
Check the balance. You will see 100 tokens in your account as the balance.

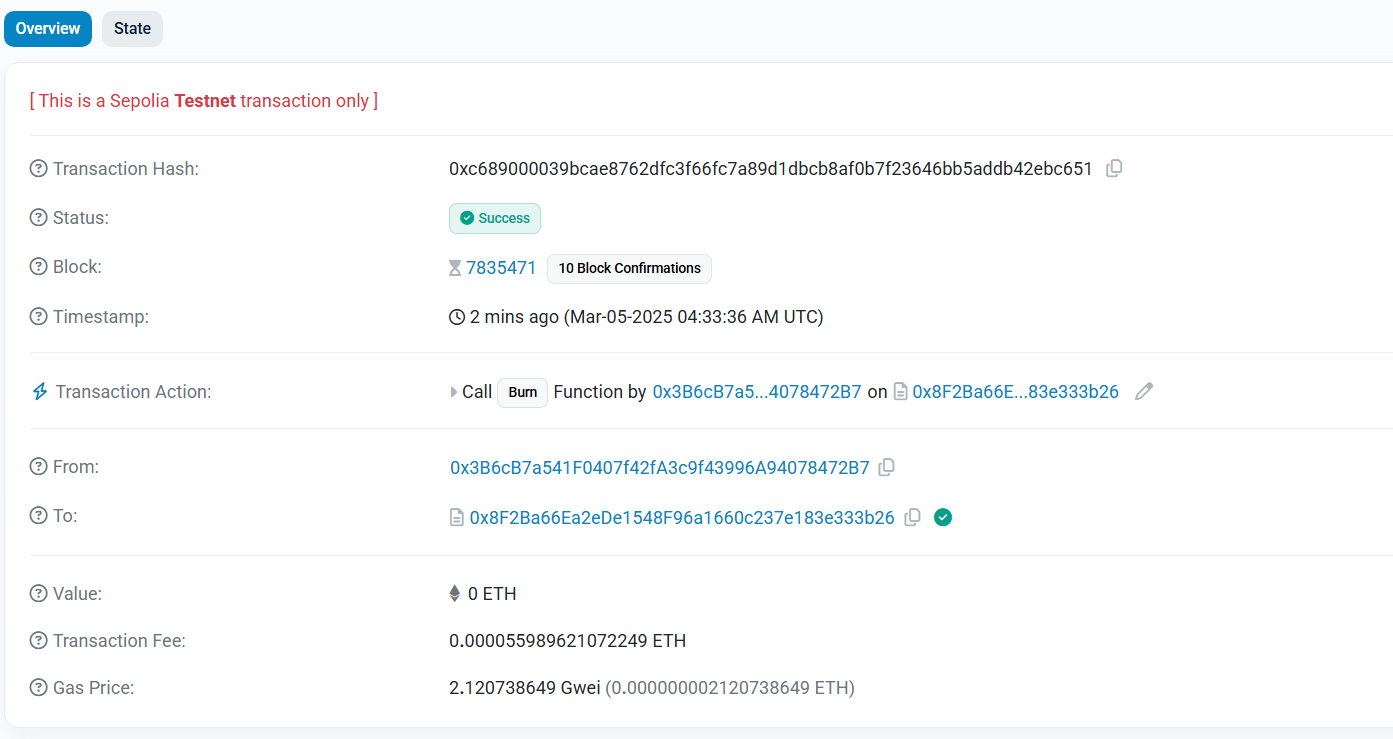


Burn 100 tokens (We had just minted 100 tokens) and you see the balance 0



View the transaction details on Etherscan





References : <https://github.com/Dev902/-ERC20-Token-Contract-with-Mint-and-Burn-Functions--/tree/main>

Assignment 3

https://github.com/andresudi/Voting-Smart-Contract

🔹 **Why use blockchain for voting?**

* Prevents **vote manipulation**.
* Ensures **transparency & security**.
* Eliminates **single points of failure**.

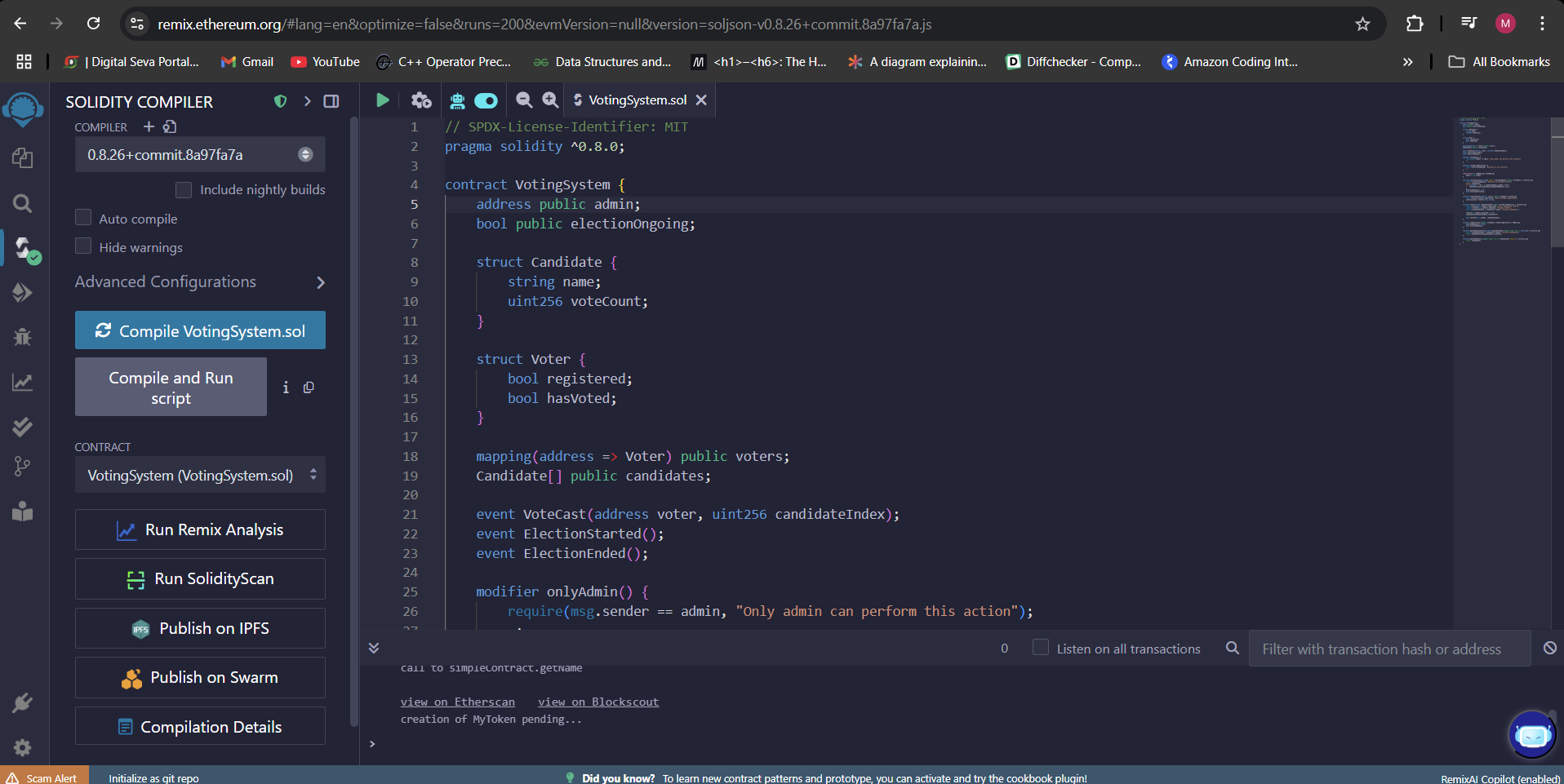
1️ Research & Design

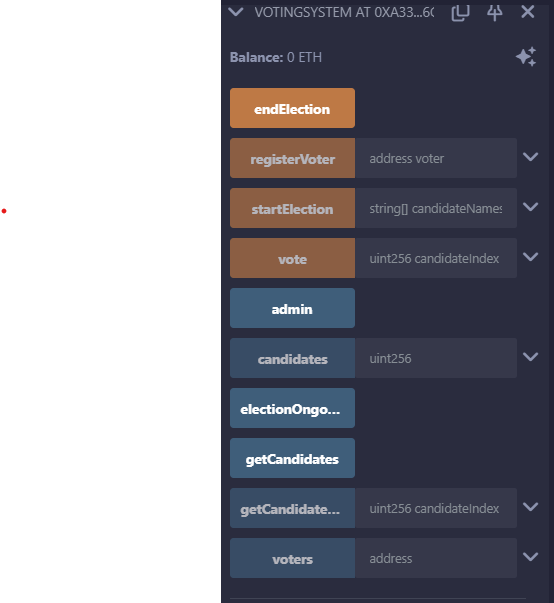
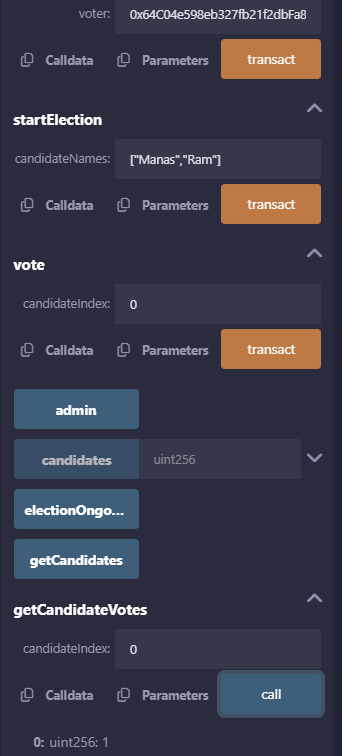
✔ **Study Blockchain-Based Voting Systems**

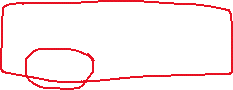
* Existing systems like **Voatz, Follow My Vote** use blockchain.
* Voting is done via **smart contracts**, ensuring immutability.

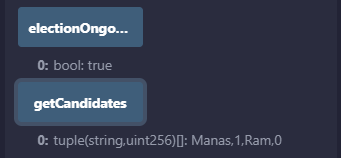
✔ **Design Voting Process**

* Only **registered voters** can vote.
* Each voter can vote **only once**.
* Votes are **publicly verifiable**, but **voter identities remain anonymous**.









**Assignment 5:** Hyperledger Fabric – Setting Up a Private Blockchain Network

This assignment focuses on setting up a **permissioned blockchain network** using **Hyperledger Fabric** and deploying a simple **chain code** (smart contract) to store and retrieve assets.

**🔹 Prerequisites**

Before installing Hyperledger Fabric, ensure your system has the following:

* **Docker** (Container runtime for Fabric)
* **Docker Compose** (For running multiple Fabric components)
* **Go** (For writing chaincode)
* **Node.js & NPM** (For SDK interaction)
* **cURL & jq** (For API testing)
* **Git** (For cloning repositories)

**🔹 Steps to Install Hyperledger Fabric**

1. **Clone the Fabric Samples Repository:**

git clone -b main https://github.com/hyperledger/fabric-samples.git

cd fabric-samples

1. **Install Fabric binaries & Docker images:**

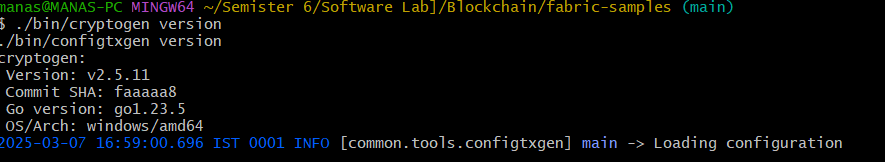
curl -sSL https://bit.ly/2ysbOFE | bash -s

This will install Fabric binaries (fabric-ca, peer, orderer, etc.).

1. **Verify Installation:**

./bin/cryptogen version

./bin/configtxgen version



**2️ Set Up a Fabric Network using Docker Compose**

Hyperledger Fabric uses Docker containers to simulate a real network.

**🔹 Start the Fabric Network**

1. **Navigate to the test network:**

cd fabric-samples/test-network

1. **Start the network:**

./network.sh up

1. **Create a channel for communication:**

./network.sh createChannel

**Smart Contract Security Audit Report**

**1. Introduction**

This report analyzes the security vulnerabilities in a real-world Ethereum smart contract and provides recommendations to mitigate potential risks. The contract chosen for this audit is a simple deposit and withdrawal contract that allows users to store and retrieve Ether. However, it contains several security flaws that could be exploited by malicious actors.

**2. Smart Contract Code**

The following Solidity contract is under review:

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");

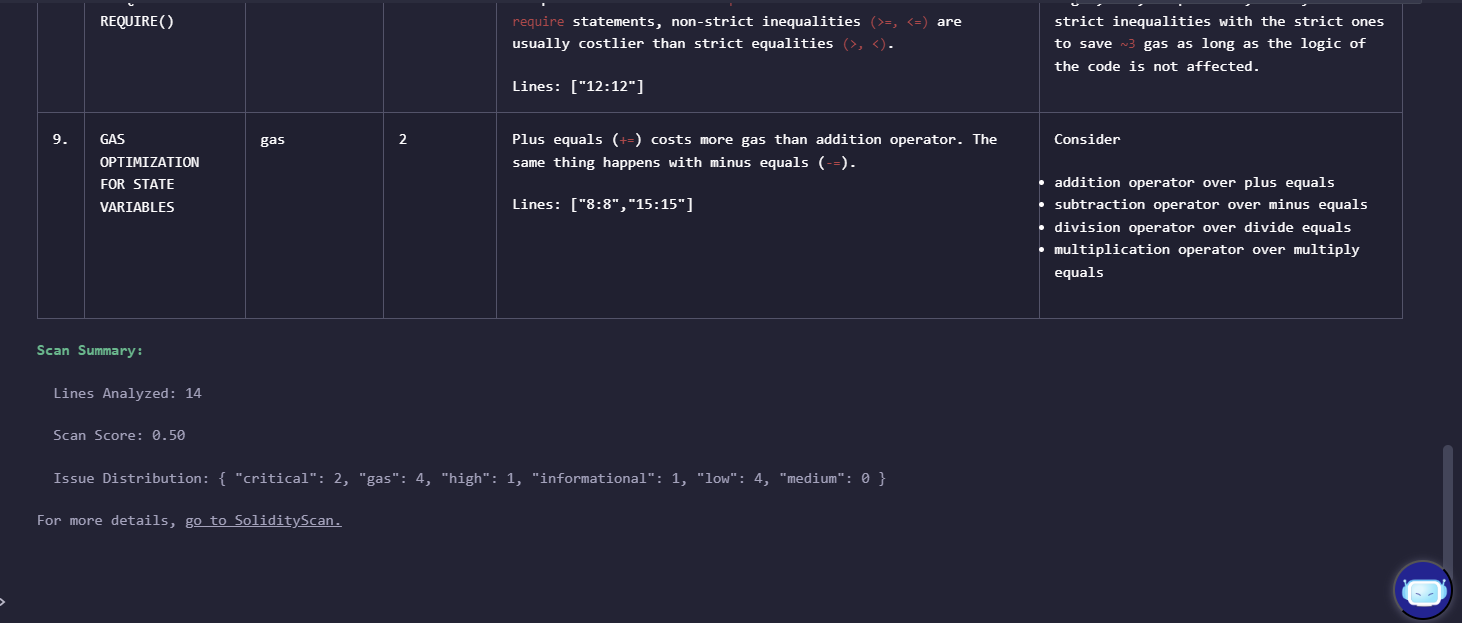
(bool success, ) = msg.sender.call{value: amount}(""); // Unsafe external call

require(success, "Transfer failed");

balances[msg.sender] -= amount;

}

}



Use the Run Solidify Scan third party application to detect potential security issues.

1. **External Call Vulnerability**: In the withdraw function, an external call (msg.sender.call{value: amount}("")) is made without proper validation or handling of errors. This can lead to reentrancy attacks if another contract attempts to withdraw Ether from a user's balance while they are still in the middle of withdrawing their own funds.
2. **Insufficient Balance Check**: While there is an require statement checking for sufficient balance, it does not handle cases where multiple contracts attempt to withdraw at once but before one has actually transferred its share back into the contract. This can lead to unexpected behavior and potential losses due to incorrect final balances being reflected in user accounts.
3. **Reentrancy**: The use of external calls (call) without proper handling for reentrant attacks makes it possible for another smart contract (which itself contains a withdraw function) to manipulate funds while inside the current contract's execution flow, potentially leading to unintended fund transfers or other malicious activities

**3. Identified Security Vulnerabilities and Fixes**

**Vulnerability 1: Reentrancy Attack**

**Issue:**

The withdraw() function transfers Ether to the caller before updating the user’s balance. This makes the contract vulnerable to a **reentrancy attack**, where a malicious contract can recursively call withdraw() before the balance is updated, allowing it to drain funds.

**Impact:**

* Attackers can exploit this to withdraw more funds than they are entitled to.
* Can lead to complete depletion of contract funds (as seen in the DAO hack).

**Fix:**

Use the **Checks-Effects-Interactions pattern**, where the balance is updated before making an external call.

**Fixed Code:**

function withdraw(uint256 amount) public {

require(balances[msg.sender] >= amount, "Insufficient balance");

balances[msg.sender] -= amount; // Update balance first

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

require(success, "Transfer failed");

}

**Vulnerability 2: Integer Overflow/Underflow**

**Issue:**

Before Solidity 0.8, integer overflows and underflows could occur, leading to miscalculations. While Solidity 0.8 provides built-in protection, older contracts remain vulnerable.

**Impact:**

* Attackers may manipulate balance calculations to withdraw excessive amounts.

**Fix:**

For older Solidity versions, use OpenZeppelin’s **SafeMath** library to prevent integer overflows.

**Fixed Code (for Solidity <0.8):**

import "@openzeppelin/contracts/utils/math/SafeMath.sol";

contract SecureContract {

using SafeMath for uint256;

function deposit(uint256 amount) public {

balances[msg.sender] = balances[msg.sender].add(amount);

}

}

**Vulnerability 3: Unprotected External Calls**

**Issue:**

The contract uses call{value: amount}("") for sending Ether, which is unsafe. If the recipient is a malicious contract, it could consume excessive gas or manipulate execution.

**Impact:**

* Funds could be lost if the transaction fails due to high gas consumption.
* Malicious contracts can re-enter execution flow.

**Fix:**

* Use transfer() or send() instead of call(), as they impose fixed gas limits.
* Implement a **withdraw pattern** where users explicitly pull funds instead of the contract pushing them.

**Fixed Code:**

function withdraw(uint256 amount) public {

require(balances[msg.sender] >= amount, "Insufficient balance");

balances[msg.sender] -= amount;

payable(msg.sender).transfer(amount); // Safer alternative

}

**4. Conclusion**

In this report, we analyzed a real Ethereum smart contract and identified three major security vulnerabilities:

1. **Reentrancy Attack** – Fixed by updating the user’s balance before making an external call.
2. **Integer Overflow/Underflow** – Addressed by using Solidity 0.8 protections or SafeMath.
3. **Unprotected External Calls** – Mitigated by using safer transfer methods and the withdraw pattern.

By applying these security best practices, we can ensure that Ethereum smart contracts remain secure against common attack vectors.



Reference : https://www.quicknode.com/guides/ethereum-development/smart-contracts/common-solidity-vulnerabilities-on-ethereum

**Building a Simple Cross-Chain Token Bridge (Ethereum ↔ Polygon)**

A **cross-chain token bridge** allows users to transfer tokens between two different blockchain networks, such as **Ethereum** and **Polygon**.

**Step 1: Understanding How Cross-Chain Bridges Work**

A bridge typically works using these steps:

1. **Lock Tokens**: A smart contract locks tokens on the source blockchain.
2. **Generate Proof**: A relayer or an oracle confirms the locked amount.
3. **Mint Tokens**: Equivalent tokens are minted on the destination blockchain.
4. **Burn & Release**: When users move tokens back, the destination tokens are burned, and the original ones are released.

**Step 2: Tools & Technologies Required**

* **Solidity**: Smart contract programming language.
* **Hardhat/Remix**: Development & deployment environment.
* **Ethereum & Polygon Testnets**: Sepolia (Ethereum) and Mumbai (Polygon).
* **Metamask**: For deploying and testing transactions.
* **Chainlink CCIP / LayerZero / Axelar**: Cross-chain communication protocols (optional but recommended).

## **Step 3: Smart Contract for Token Locking (Ethereum Side)**

This contract locks tokens before transferring them to Polygon.

✅ **What This Does?**

* Users deposit tokens into this contract, which locks them.
* A relayer (or smart contract on Polygon) can later request to release them

## **Step 4: Smart Contract for Token Minting (Polygon Side)**

This contract **mints new tokens** equivalent to the locked amount.

✅ **What This Does?**

* The **mintTokens()** function allows an admin (relayer) to mint tokens.
* The **burnTokens()** function burns the minted tokens when bridging back to Ethereum.

**Step 5: Deploying Contracts**

1. **Deploy the EthereumBridge contract** on Sepolia (Ethereum Testnet).
2. **Deploy the PolygonToken contract** on Mumbai (Polygon Testnet).

Use **Remix IDE** or **Hardhat** for deployment.

**Step 6: Testing the Bridge**

1. **Deposit ERC-20 tokens into the EthereumBridge contract.**
2. **Trigger minting on the Polygon contract manually (or via a relayer).**
3. **Verify tokens appear in the Polygon wallet.**
4. **Burn tokens on Polygon and release them back on Ethereum.**

**Step 7: Automating the Bridge**

Instead of manually calling unlockTokens() and mintTokens(), we can use:

1. **Chainlink CCIP**: A decentralized oracle network for cross-chain messages.
2. **LayerZero or Axelar**: For automatic message passing between Ethereum and Polygon.

