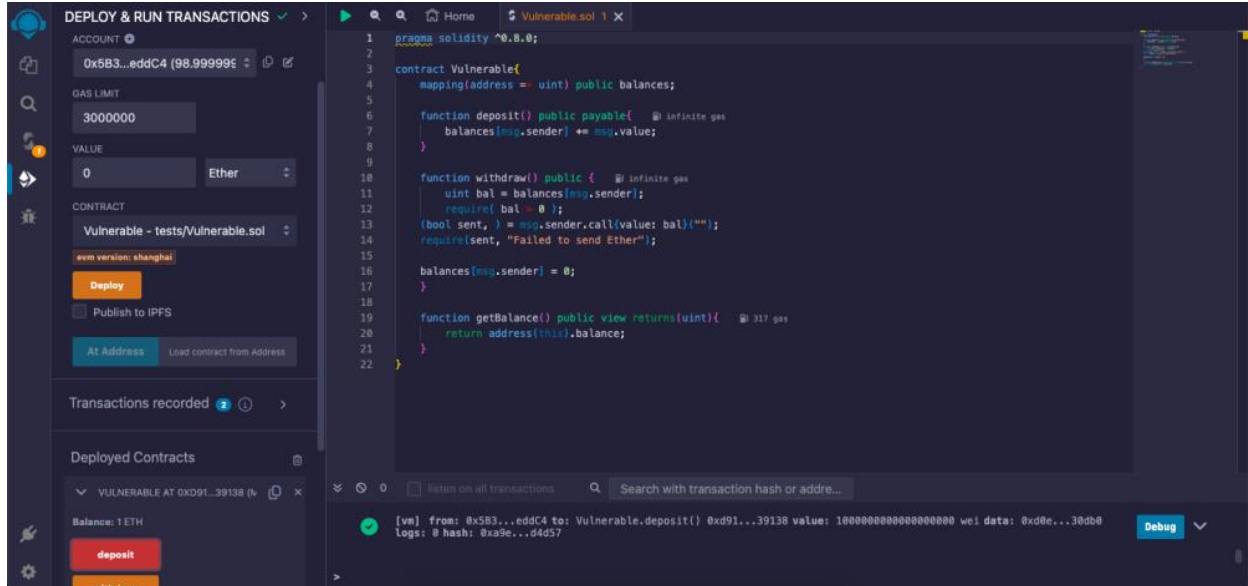


Reentrancy Vulnerable Contract Documentation

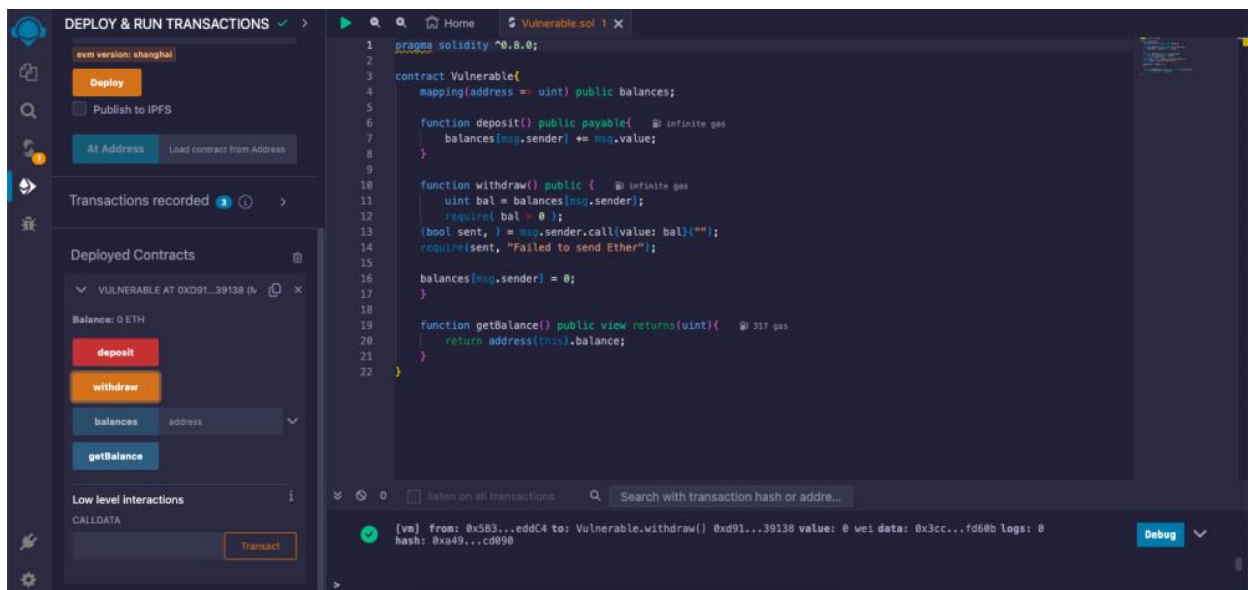
In this documentation, I will take you through the testing of a contract that has a reentrancy vulnerability in it and the associated malicious contract that successfully exploits the vulnerability. Additionally, I will show you two different mitigation techniques and the successful mitigation of the vulnerability.

1. Vulnerable Contract deposit Function Test



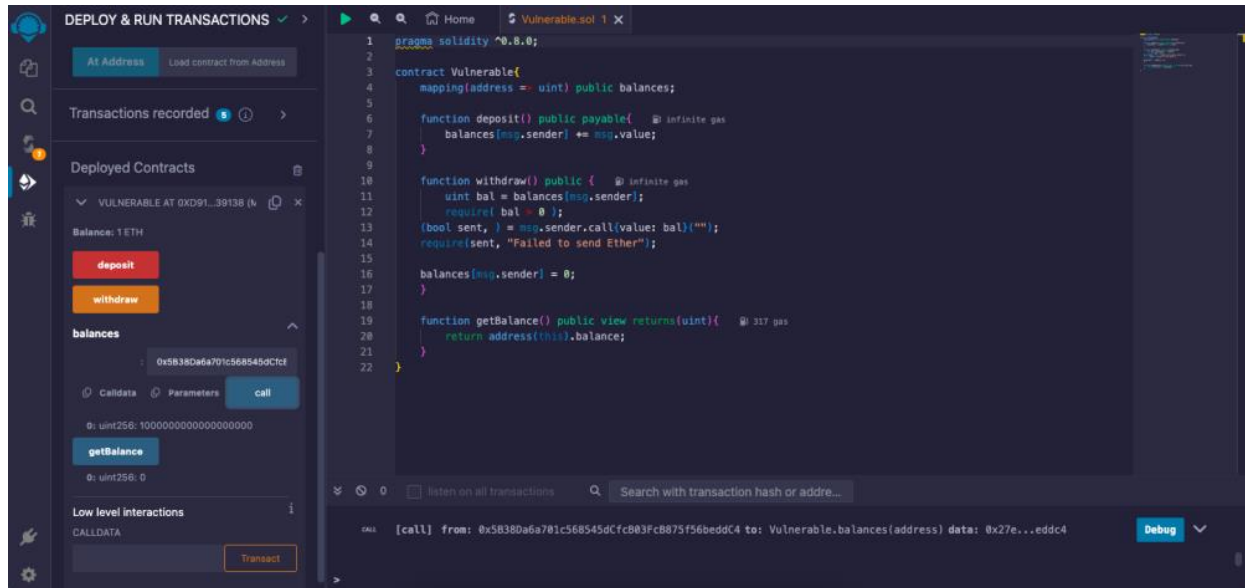
In this screenshot, you can see the vulnerable contract that has been written, compiled, and deployed within a Remix VM. I tested the Deposit function by entering 1 Ether as the value in the Value section and clicking the *deposit* function under the contract. The contract's balance updated to show 1 ETH and a green checkmark appeared next to the transaction in the log.

2. Vulnerable Contract withdraw Function Test



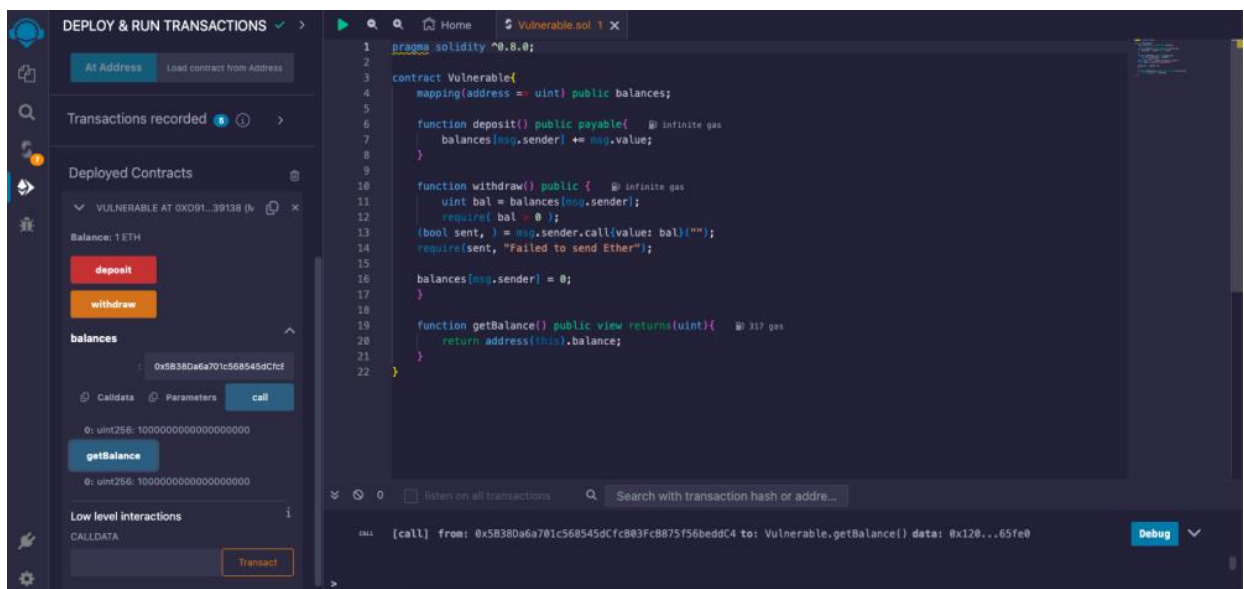
Here I tested the *withdraw* function by clicking the withdraw button under the contract. The contract's balance updated to show 0 ETH and a green checkmark appeared next to the transaction in the log.

3. Vulnerable Contract balances Variable Test



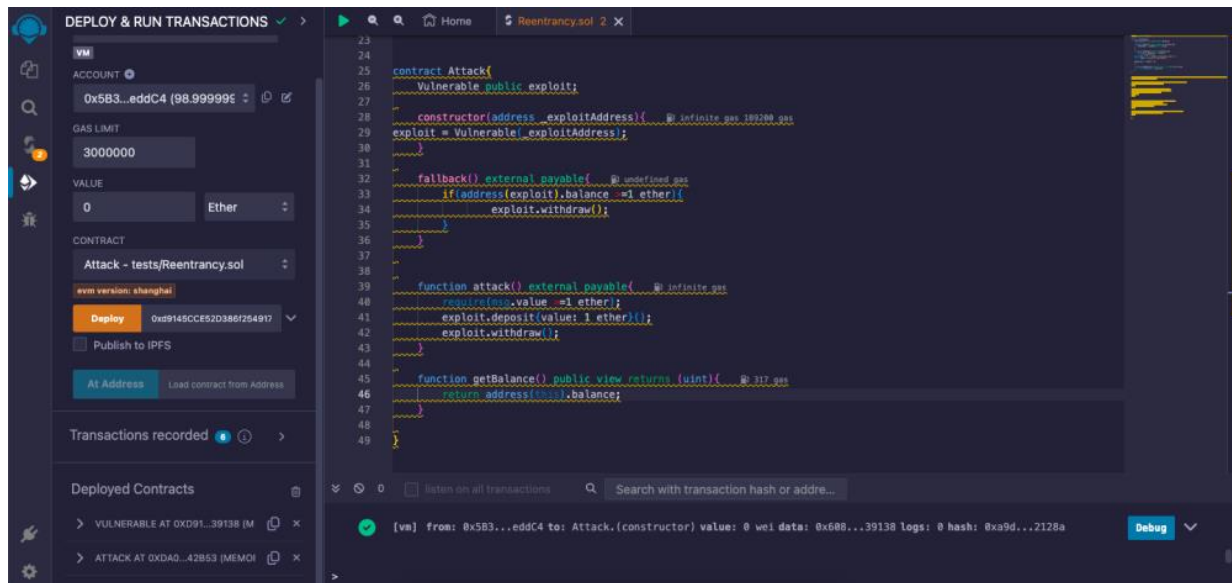
To test this function, I refunded the contract then copied and pasted the depositing address into the value field under the balances variable then I pressed the *call* function. The function returned a value of 1 ETH (written in Wei).

4. Vulnerable Contract getBalance Function Test



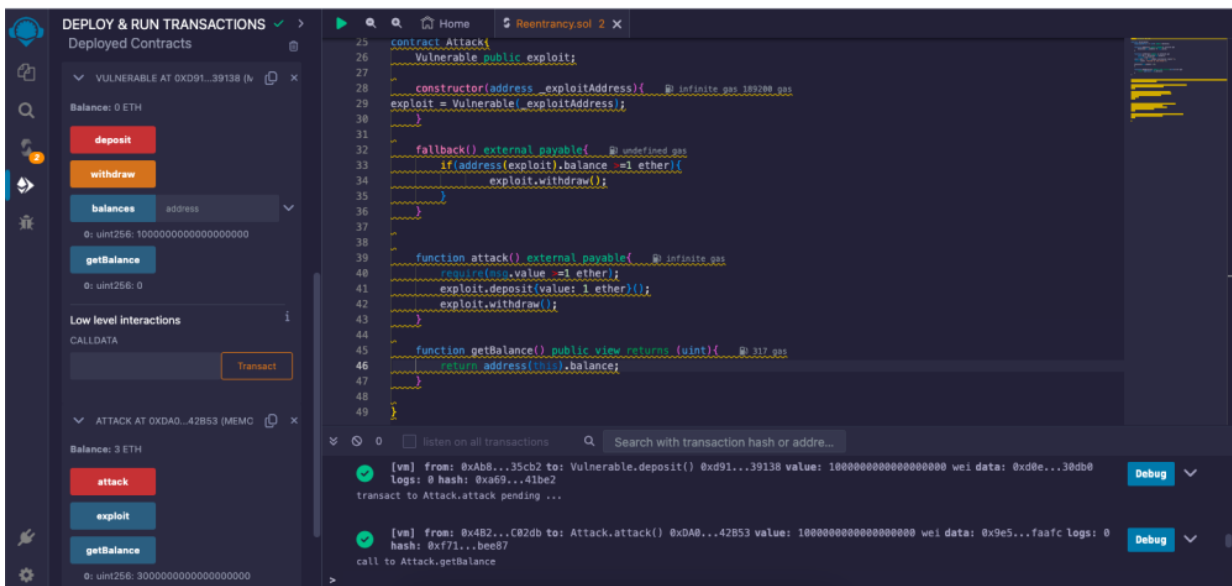
Finally, to test the *getBalance* function, I simply clicked the *getBalance* button in the contract and it returned a value of 1 ETH (written in Wei) because the contract contained 1 ETH at that point in time.

5. Deploying the Malicious Contract



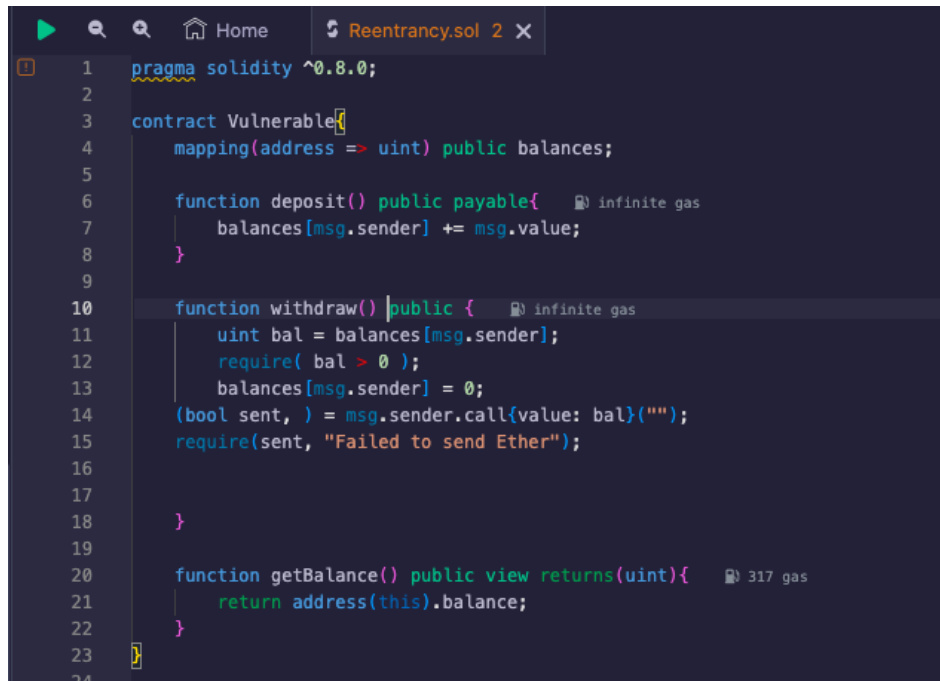
In this screenshot, you can see the malicious contract has been written, compiled, and deployed within the same Remix VM using the address of the vulnerable contract.

6. Exploiting the Vulnerable Contract



So, I proceeded to fund the vulnerable contract with 2 Ether from two different addresses. After that, I switched to a third address, entered 1 Ether as the value in the Value section (not shown) and then clicked the *attack* function under the malicious contract. This caused the malicious contract to first deposit the 1 Ether into the vulnerable contract and then immediately call the vulnerable contract's *withdraw* function. This caused all the Ether in the vulnerable contract to be transferred to the malicious contract. This is verified by the 3 ETH balance in the malicious contract, the 0 ETH balance in the vulnerable contract, and the two green checkmarks next to the respective transactions in the log.

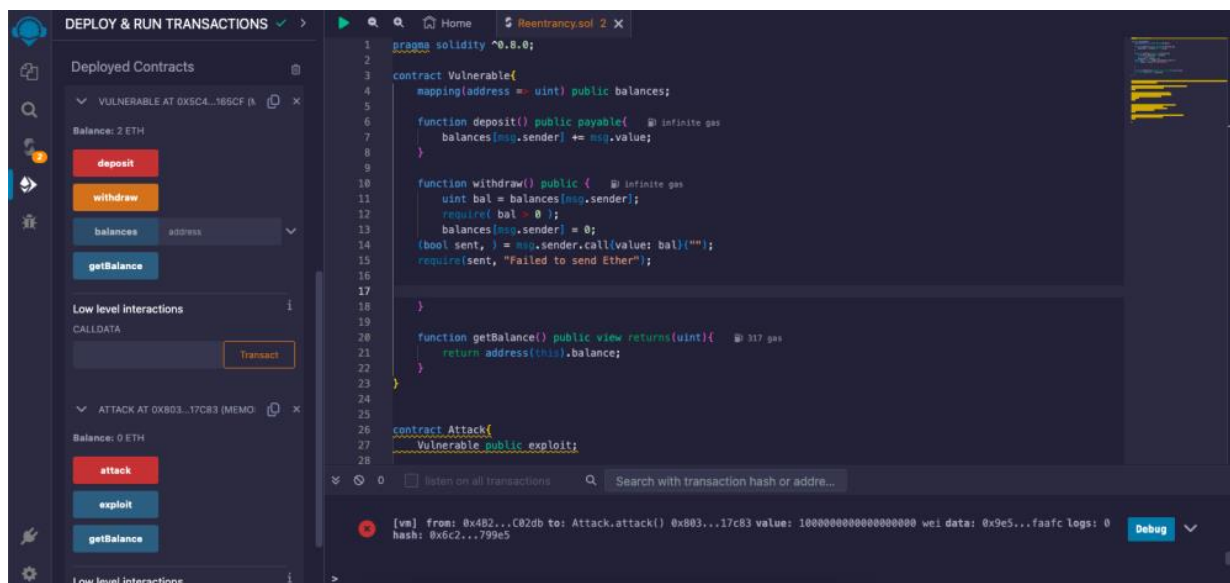
7. First Mitigation Strategy



```
1 pragma solidity ^0.8.0;
2
3 contract Vulnerable{
4     mapping(address => uint) public balances;
5
6     function deposit() public payable{ @infinite gas
7         balances[msg.sender] += msg.value;
8     }
9
10    function withdraw() public { @infinite gas
11        uint bal = balances[msg.sender];
12        require( bal > 0 );
13        balances[msg.sender] = 0;
14        (bool sent, ) = msg.sender.call{value: bal}("");
15        require(sent, "Failed to send Ether");
16    }
17
18
19
20    function getBalance() public view returns(uint){ @317 gas
21        return address(this).balance;
22    }
23
24 }
```

This is an updated version of the vulnerable contract that no longer has the reentrancy vulnerability within it. This strategy was achieved by moving the "balances[msg.sender] = 0;" code in line 16 above the "(bool sent,) = msg.sender.call{value: bal}("");" code on line 13, which ensures that the balance of the contract is updated to 0 before payment is made to the external contract.

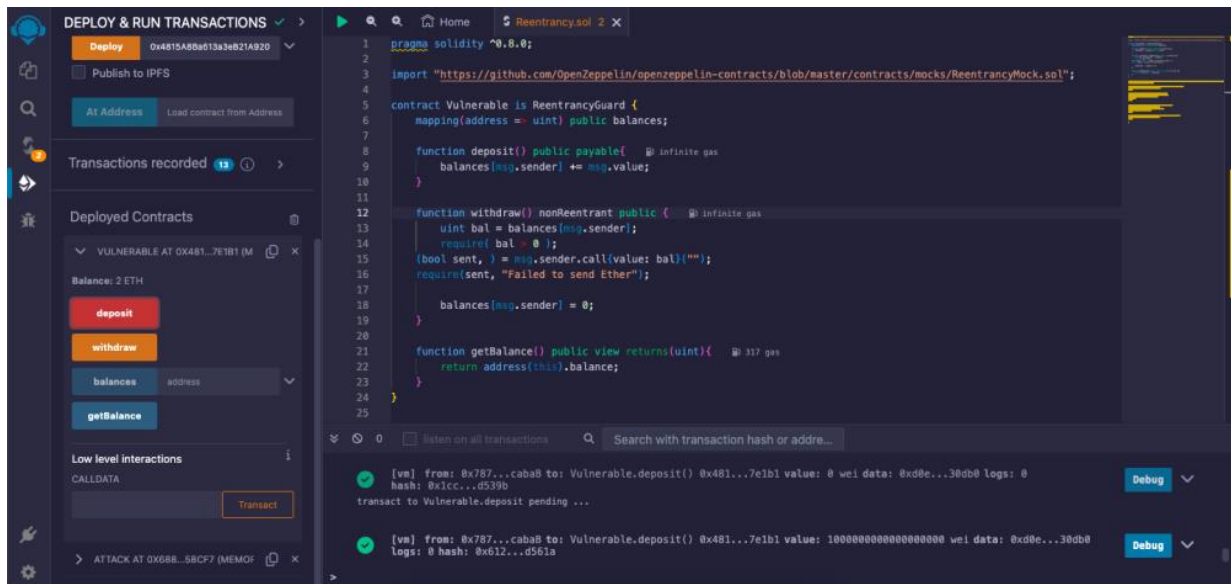
8. First Mitigation Strategy Proof



The screenshot shows the Remix IDE interface. On the left, the 'DEPLOY & RUN TRANSACTIONS' panel displays two contracts: 'VULNERABLE AT 0X5C4...185CF (A)' and 'ATTACK AT 0XB03...17C83 (MEMO)'. The 'Vulnerable' contract has a balance of 2 ETH, and the 'Attack' contract has a balance of 0 ETH. The 'Vulnerable' contract has buttons for 'deposit', 'withdraw', 'balances', and 'getBalance'. The 'Attack' contract has buttons for 'attack', 'exploit', and 'getBalance'. On the right, the Solidity code editor shows the 'Vulnerable' contract code, which is the same as in the previous image. Below the code editor, the 'Log' panel shows a transaction from '0x4B2...C82db' to 'Attack.attack()' with a value of '1000000000000000000 wei' and a data field '0x9e5...fa5c'. A red X icon is next to the transaction, indicating it failed. The 'Debug' button is visible next to the log entry.

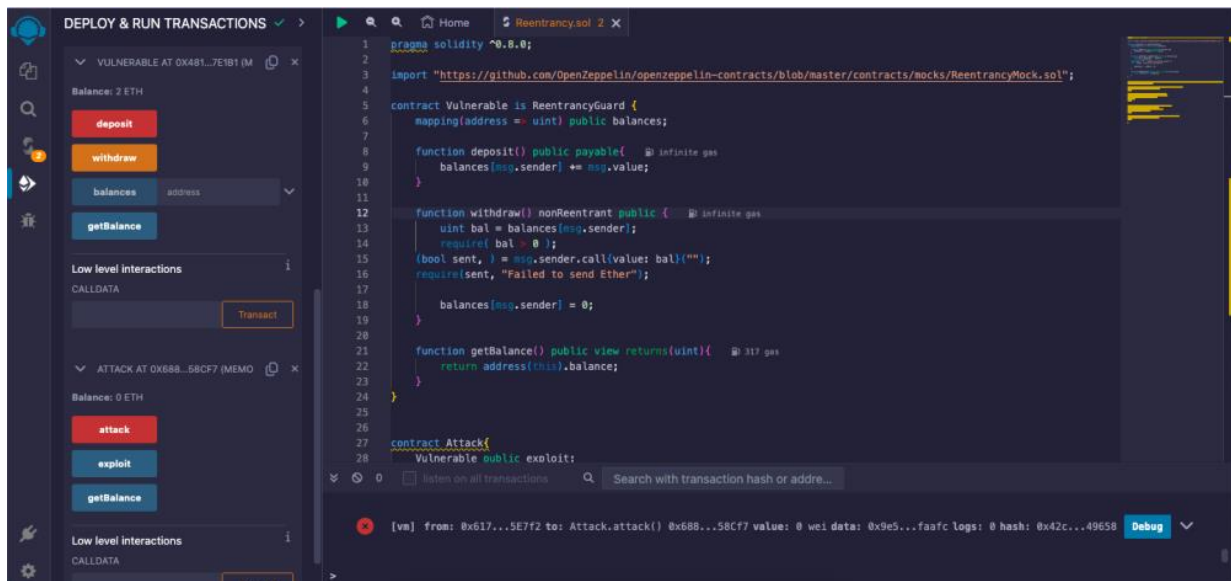
To prove that this strategy works, I redeployed both contracts and funded the now not vulnerable contract with 2 ETH, then I entered 1 Ether as the value in the Value section (not shown) and clicked the *attack* function under the malicious contract. The attack did not execute and that is verified by the 2 ETH balance remaining in the now not vulnerable contract and the 0 ETH balance in the malicious contract. Additionally, a red X appeared next to the attempted transaction in the log.

9. Second Mitigation Strategy



Here you can see the updated contract that includes Reentrancy Guard. The contracts have been redeployed and the “Vulnerable” contract has been funded with 2 ETH.

10. Second Mitigation Strategy Proof



I then entered 1 Ether as the value in the Value section (not shown) and clicked the *attack* function under the malicious contract. The attack did not execute and that is verified by the 2 ETH balance remaining in the “Vulnerable” contract and the 0 ETH balance in the malicious contract. Additionally, a red X appeared next to the attempted transaction in the log.