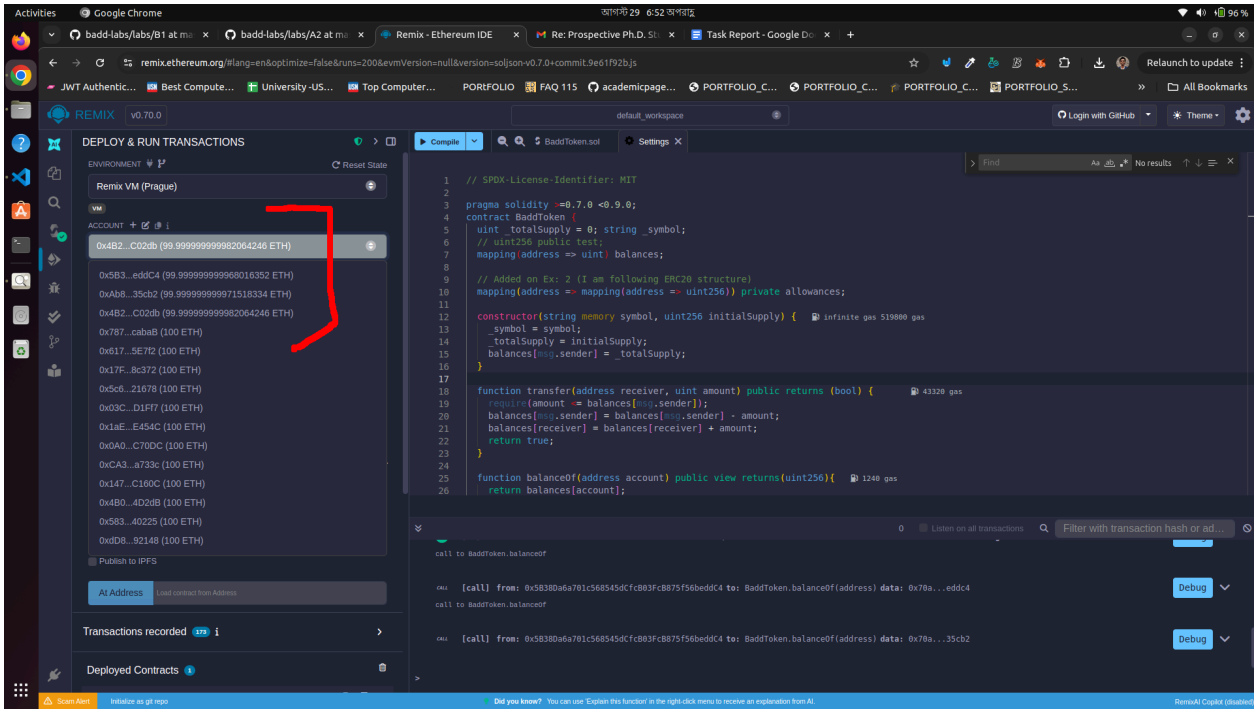
Solution (Exercise)

By
Md Yeasin Ali
Prospective PhD Student

Prerequisite:

All of the tasks are executed on Remix IDE as described in the instructions. We used three EOA accounts for all the tests. We consider:
Alice = 0x5B38Da6a701c568545dCfcB03FcB875f56bed**dC4**
Bob = 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835**cb2**
A third account, say Charlie = 0x4B20993Bc481177ec7E8f571ceCaE8A9e22C0**2db**

During the test, these accounts can be visible in the top-left of the screenshots - 1.

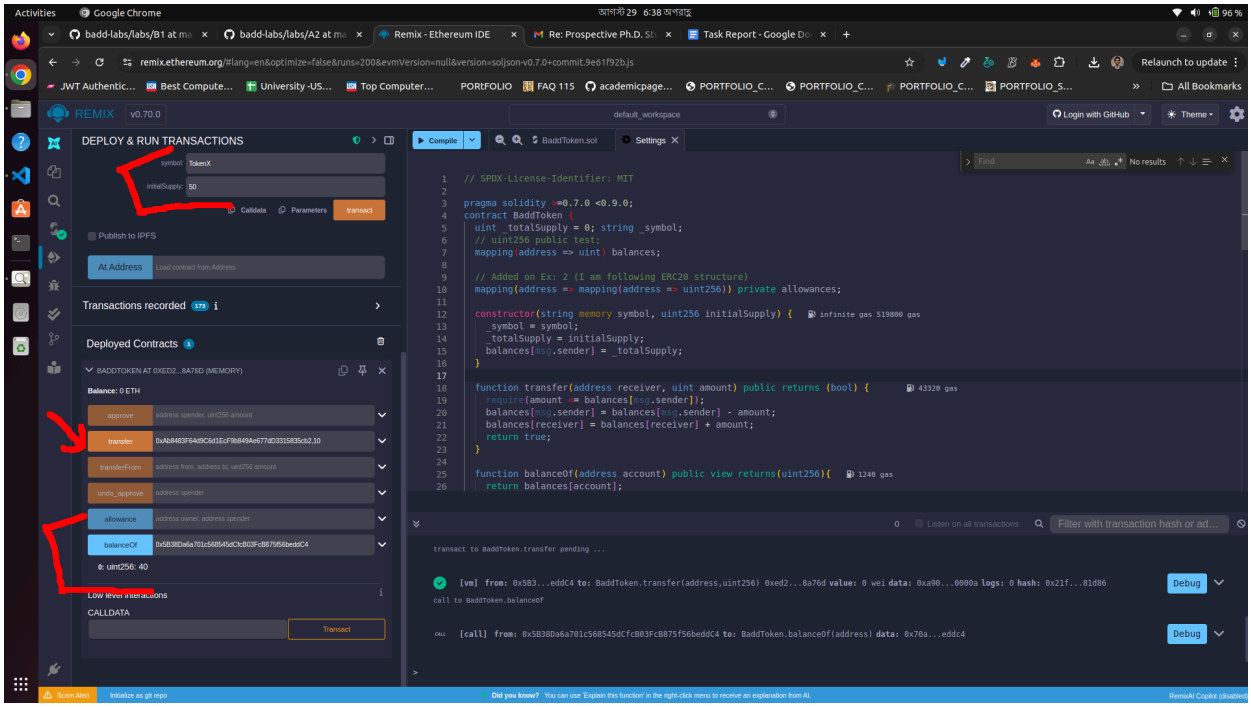


Screenshot 1

Compiler: The compiler we used was 0.8.20. However, for Exercise 4 (security analysis), it is mandatory to run a different compiler (Discussed later in this report).

Exercise 1 - Execute ERC20 token transfer

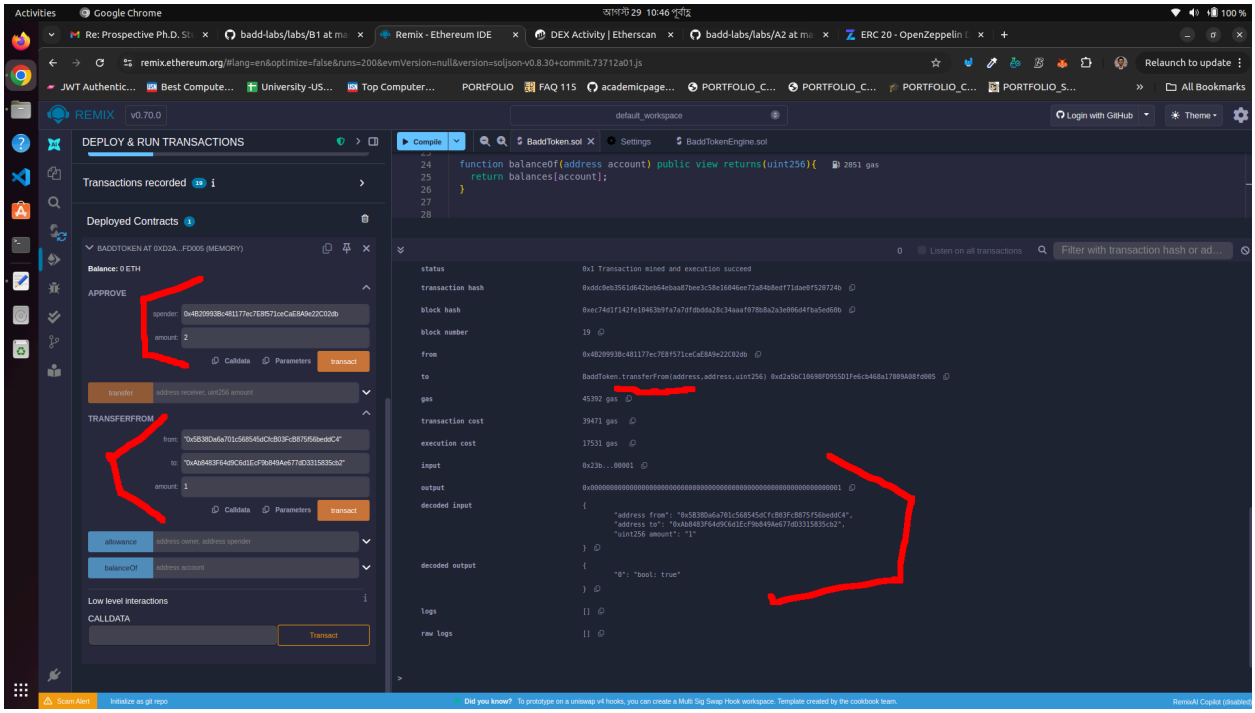
The first task is to transfer an amount of token from one account to another. We deploy an instance of the contract with initial supply for Alice = 50 token. Next, using the transfer function, 10 TokenX is transferred to Bob. This deducts 10 Tokens from Alice, making it 40. (Screenshot 2).



Screenshot 2

Exercise 2

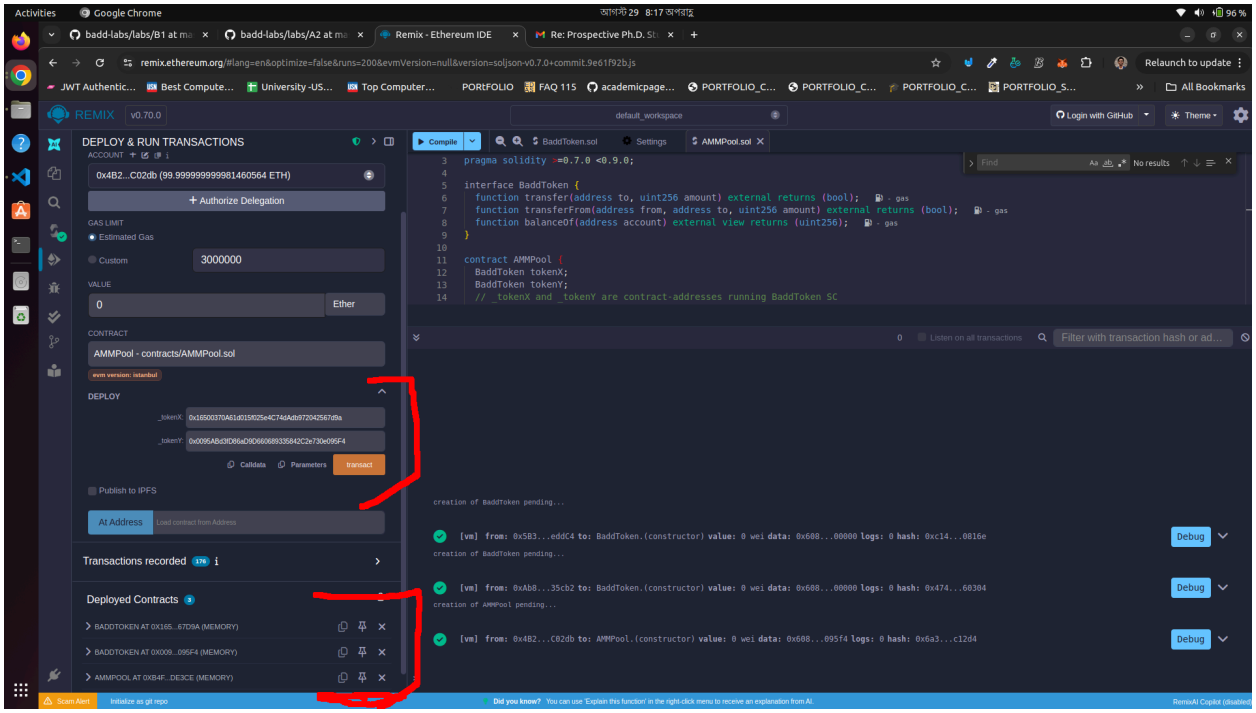
In exercise 2, Alice first **approves** Charlie's EOA account to delegate some amount to transact. Screenshot 3, shows we approve Charlie's account with 2 TokenXs, and execute the “transferFrom” function to transfer 1 token from Alice to Bob, via Charlie's EOA account.



Screenshot 3

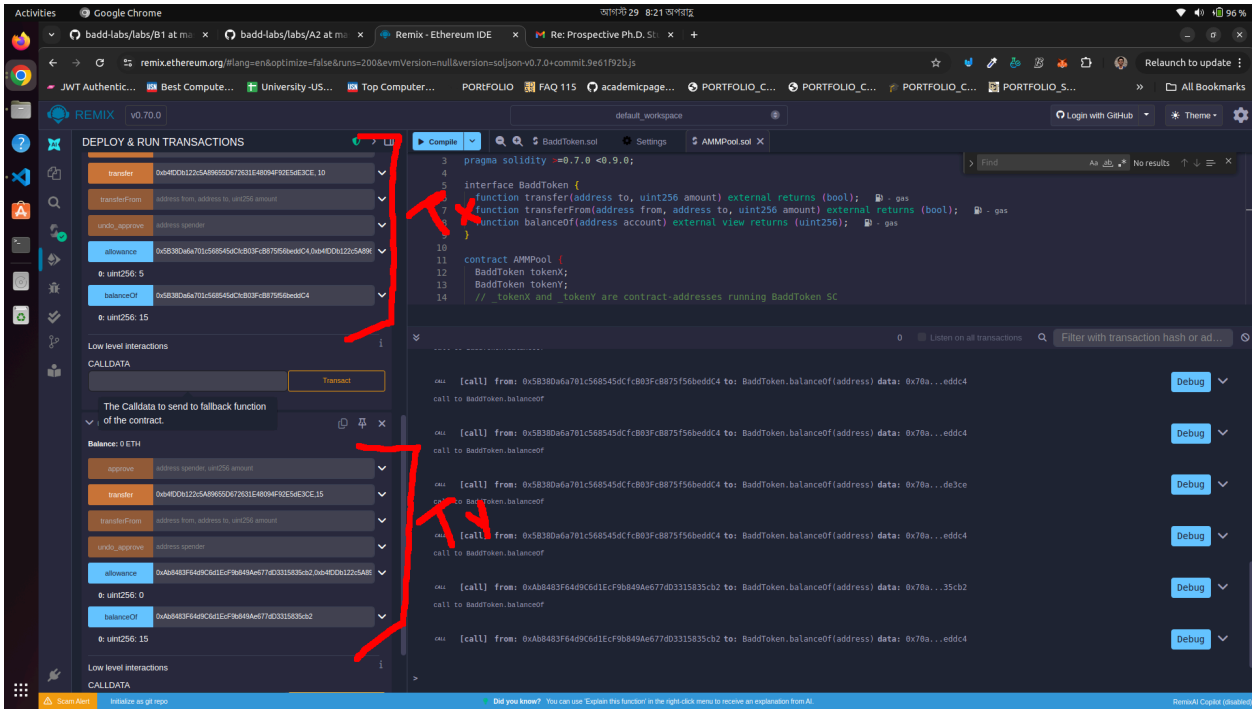
Exercise 3

In this part, a new contract called **AMMPool** is created that works as a liquidity pool for tokens. First, using the **BaddToken** contract, we create TokenX from Alice's EOA address and TokenY from Bob's EOA address. Next, we deploy the AMMPool contract that takes the contract address of TokenX and TokenY as constructor parameters (Screenshot 4).



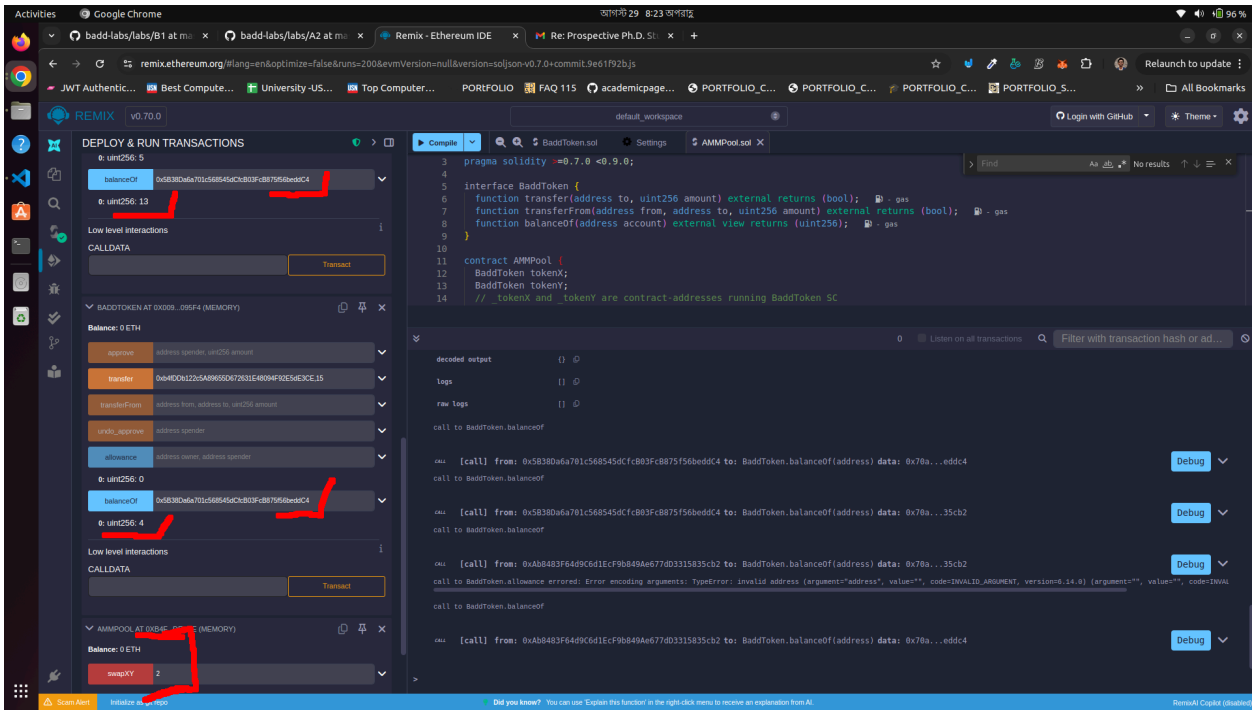
Screenshot 4

Once AMMPool is deployed, we **approve** its contract address from Alice's EOA address with 10 TokenX. However, initially the liquidity of AMMPool is zero, therefore we need to fund it with TokenX and TokenY to make it functional. Without initial token holdings, it gets reverted.



Screenshot 5

So we transferred tokens from both TokenX and TokenY to the AMMPool contract running the "**transfer**" function manually from remix IDE. As shown in screenshot-5, we transferred 10 TokenX and 15 TokenY to the AMMPool contract. After transferring, the balance in Alice's account (address 0X5….**dC4**) remains 15 TokenX (Screenshot-5). Next, we execute **swapXY** function using Alice's EOA address and swap 2 token. Therefore, it successfully transfers 2 TokenX from Alice's wallet and in exchange, receives 4 TokenY via AMMPool (Screenshot-6).

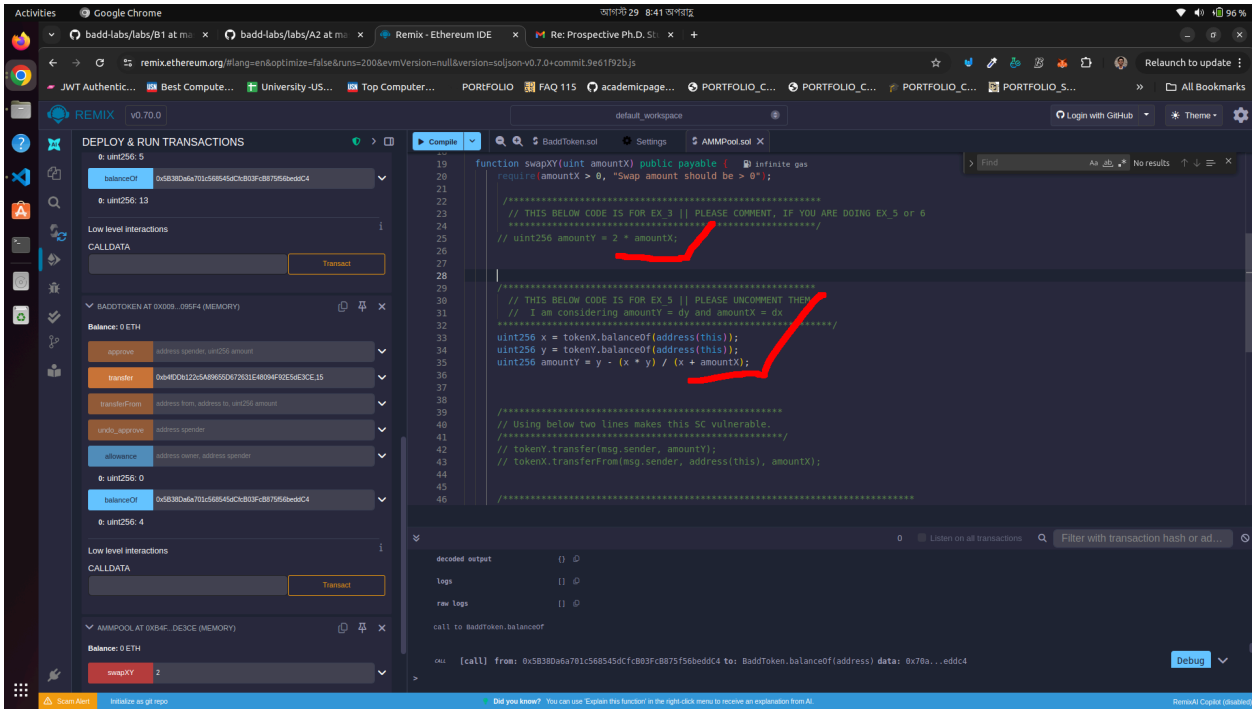


Screenshot 6

Exercise 4 : We discuss it at the end of this report.

Exercise 5

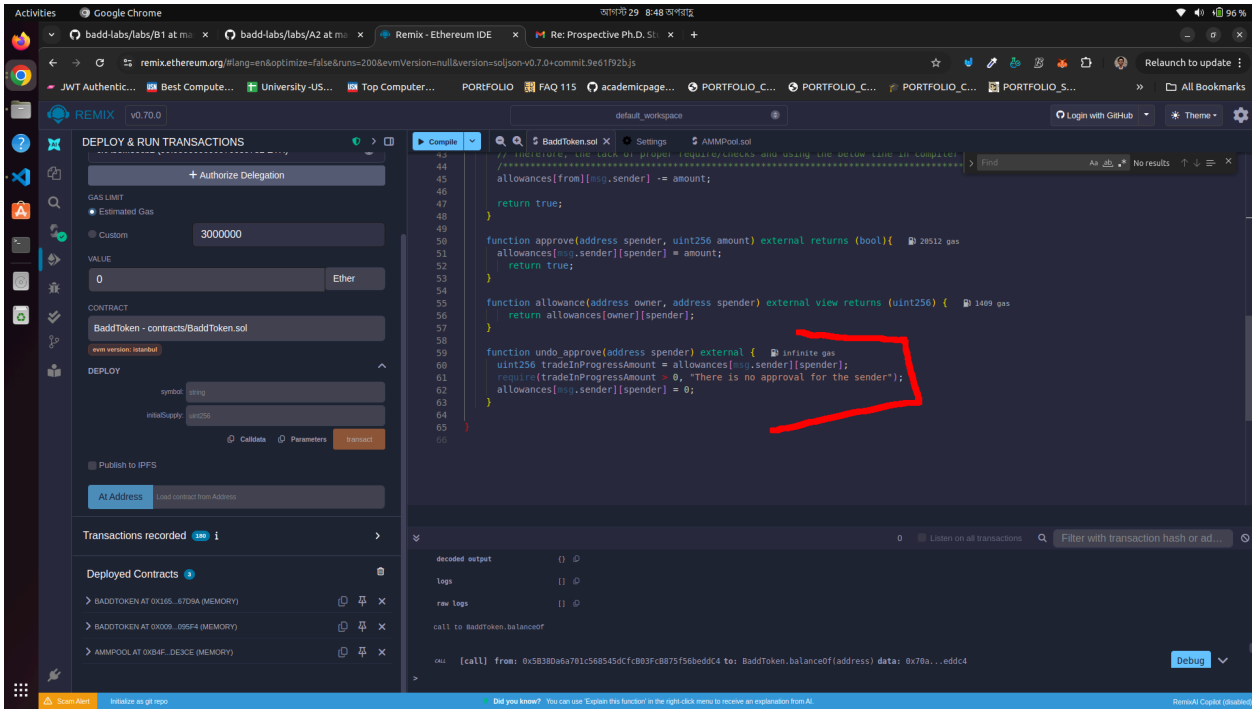
Exercise 5 was similar to 3. However, the calculation has been updated as follow:



Screenshot 7

Exercise 6

In exercise 6, the following function is added to **undo_approve** of trade-in-progress tokens.



Screenshot 8

Here, we reduce the allowances nested mapping back to “0” which worked as expected. We could use ***allowance[msg.sender][spender] = tradeInProgressAmount***, however, it could lead to overflow or underflow in different conditions.

Exercise 4 - Security Analysis (Bonus Exercise)

In this exercise, one vulnerability discusses Bob's capability to swap even without approving AMMPool. The second topic covers front-running attacks.

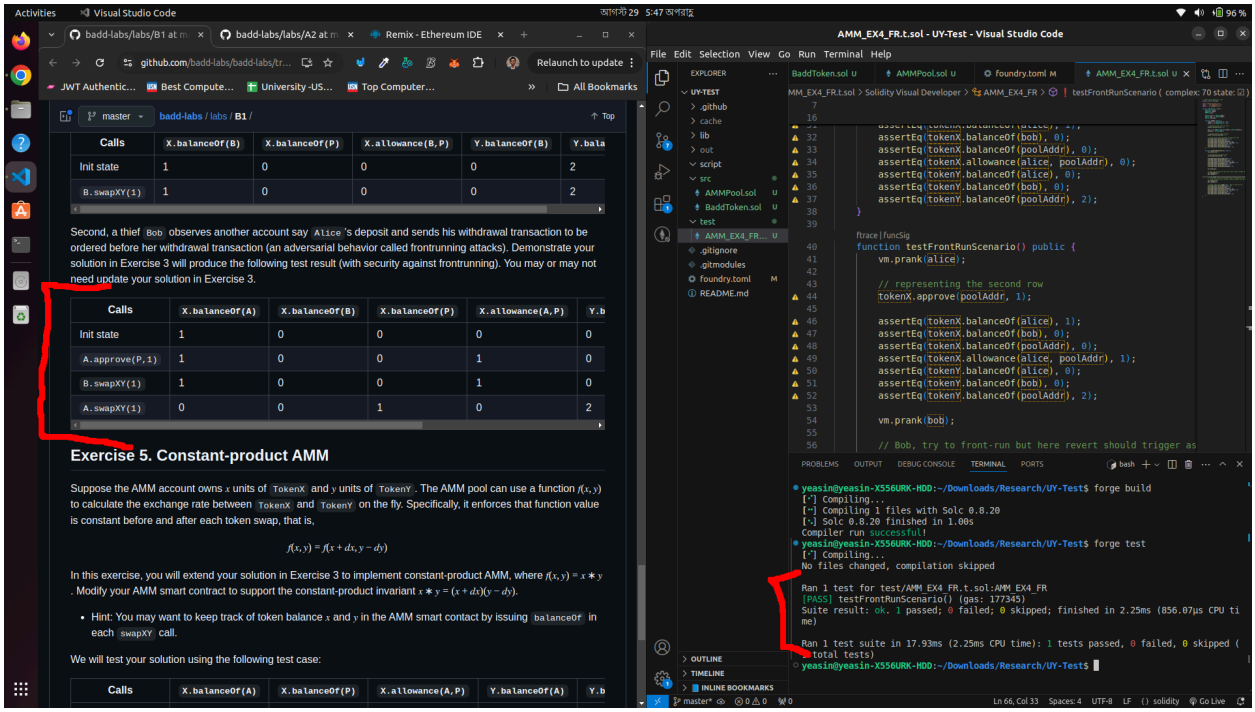
First Vulnerability

We answered this at the very last, when all the other exercises were complete. However, after completing the all task, we observed that we already wrote different sanity checking using the ***require*** statement and that prevents the vulnerability mentioned in exercise 4. However, we have modified the secured code and provided a vulnerable version too (*Please see the vulnerable code folder*). In that vulnerable version, we removed multiple ***require*** statements that could overflow/underflow. However, since solidity compiler version 0.8+ by default prevents those, the vulnerable code must be run in compiler version lesser than 0.8, as they do not prevent accidental overflow and underflow by default. We have added some comments in the contract to highlight which part of the contract can make it vulnerable.

NOTE: Therefore, to verify the vulnerable version, it is mandatory to use a compiler version less than 0.8 to see how Bob swaps tokens without even approving.

Front Running

For the second part of the front-running attack, we demonstrate that our code can prevents the front-running attack and successfully pass the test cases. However, for this we execute a test in **foundry** as we believe it would be a better way to demonstrate how it satisfies the cases. The test code has already been provided and also can be shown in screenshot 9.



Screenshot 9