Introduction to Smart Contract Exploitation

Presenter: Josselin Feist, josselin@trailofbits.com

Download the virtual machine.
The login/password are `grehack/grehack`.
To deploy the exercises, run:

```
/home/grehack/Desktop/start_exercises.sh
```

You can re-start the exercises by running again the script.

Alternatively, follow the installation instructions for a local installation.

Need help? Slack: https://empireslacking.herokuapp.com/ #ethereum

Prerequisites

Ganache

Ganache allows to prototype and to test smart contracts on a local blockchain.
The local blockchain validates instantaneously the transactions, and allow to have pre-mined ethers.

The Ganache GUI lets you see the transactions and the blocked mined.

Geth

Geth is an ethereum client.
With geth, you can:

- Mine,
- Transfer funds
- Create a contract
- Sends transactions

Solc

Solc is the Solidity compiler.

Exercise 1: Robbing the Bank

In this exercise we will see how to interact and exploit the following contract:

```
contract Bank {
   mapping (address => uint) tokens;

   // Return the balance of the user
   function getBalance(address user) view public returns(uint) {
       return tokens[user];
   }

   // Buy tokens
   function buy() payable public{
       tokens[msg.sender] += msg.value;
   }

   // Exchange the token for ether
   function withdraw() public{
       // Retrieve the user balance
       uint balance = tokens[msg.sender];
       // Send the balance to 0
       tokens[msg.sender] = 0;
       // Send the ether equivalent of the balance
       msg.sender.transfer(balance);
   }

   // Send tokens to another user
   function transfer(address to, uint amount) public{
       tokens[to] += amount;
       tokens[msg.sender] -= amount;
   }
}
```

Figure1: Bank.sol

The contract allows anyone to buy its tokens.
It contains four functions:

- `getBalance`: returns the tokens balance of the user
- `buy`: buy tokens with ether
- `withdraw`: convert back the tokens to ether
- `transfer`: transfer the tokens to another user

Interacting with the contract

Our goal will be to buy 1000 tokens by calling the function `buy` with 1000 wei (1 wei is 1e-18 ether).

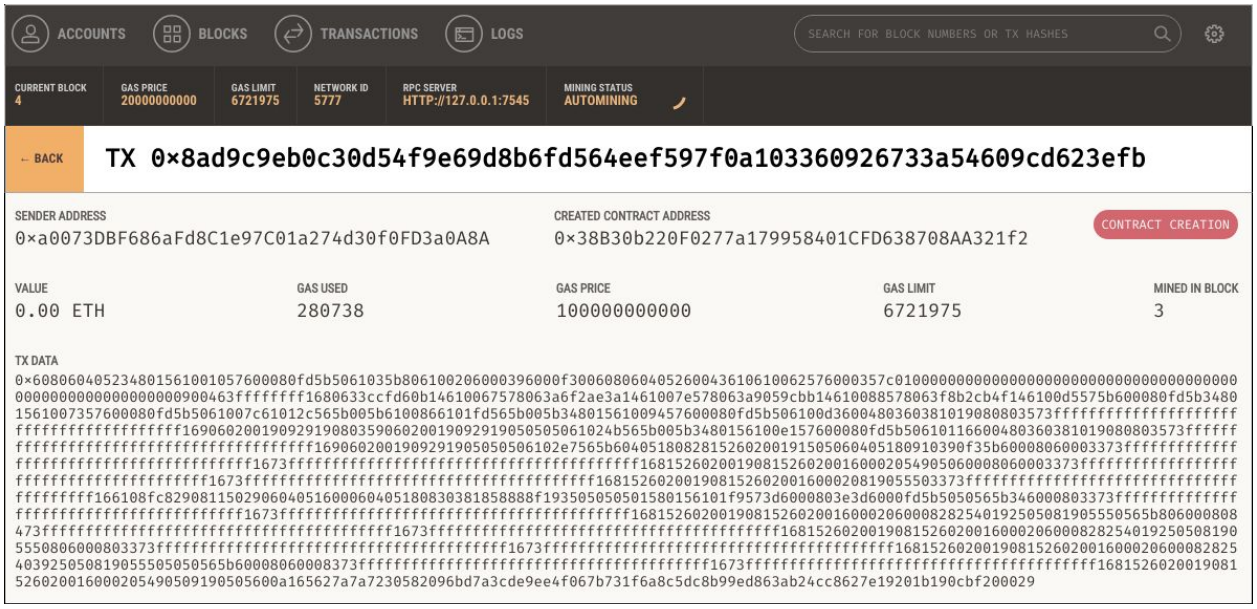
The first thing to do is to attach geth to the local blockchain:

```
$ geth attach http://127.0.0.1:7545
```

This opens a terminal to interact with the blockchain.

The contract is deployed at the address 0x38B30b220F0277a179958401CFD638708AA321f2.

You can see the contract creation at the transaction 0x8ad9c9eb0c30d54f9e69d8b6fd564eef597f0a103360926733a54609cd623efb on ganache:



CURRENT BLOCK 4 | GAS PRICE 20000000000 | GAS LIMIT 6721975 | NETWORK ID 5777 | RPC SERVER HTTP://127.0.0.1:7545 | MINING STATUS AUTOMINING

TX 0×8ad9c9eb0c30d54f9e69d8b6fd564eef597f0a103360926733a54609cd623efb

SENDER ADDRESS
0×a0073DBF686aFd8C1e97C01a274d30f0FD3a0A8A

CREATED CONTRACT ADDRESS
0×38B30b220F0277a179958401CFD638708AA321f2

VALUE
0.00 ETH

GAS USED
280738

GAS PRICE
100000000000

GAS LIMIT
6721975

MINED IN BLOCK
3

TX DATA

```
0×608060405234801561001057600080fd5b5061035b806100206000396000f300608060405260043610610062576000357c0100000000000000000000000000000000000000000
00000000000000000000900463ffffffff1680633ccfd60b14610067578063a6f2ae3a1461007e578063a9059cbb14610088578063f8b2cb4f146100d5575b600080fd5b3480
1561007357600080fd5b5061007c61012c565b005b6100866101fd565b005b34801561009457600080fd5b506100d3600480360381019080803573ffffffffffffffffffff
ffffffffffffffffff16906020019092919080359060200190929190505050610 24b565b005b3480156100e157600080fd5b50610116600480360381019080803573ffffff
ffffffffffffffffffffffffffffffffff1690602001909291905050506102e7565b6040518082815260200191505060405180910390f35b60008060003373ffffffffffff
ffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff16815260200190815260200160002054905060008060003373ffffffffffffffffff
ffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff168152602001908152602001600020819055503373ffffffffffffffffffffffffffffffff
ffffffff166108fc8290811502906040516000604051808303818588 88f193505050501580156101f9573d6000803e3d6000fd5b5050565b346000803373ffffffffffffff
ffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff16815260200190815260200160002060008282540192505081905550565b806000808
473ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff1681526020019081526020016000206000828254019250508190
5550806000803373ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff168152602001908152602001600020600082825
40392505081905550505050565b60008060008373ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff1681526020019081
526020016000205490509190505600a165627a7a7230582096bd7a3cde9ee4f067b731f6a8c5dc8b99ed863ab24cc8627e19201b190cbf200029
```

Figure 2: Ganache GUI

To interact with the contract in geth, you need to export the contract ABI using solc:

```
      $ solc bank.sol--abi
      [{"constant":false,"inputs":[],"name":"withdraw","outputs":[],"payable"
:false,"stateMutability":"nonpayable","type":"function"},{"constant":false,"i
nputs":[],"name":"buy","outputs":[],"payable":true,"stateMutability":"payable
","type":"function"},{"constant":false,"inputs":[{"name":"to","type":"address
"},{"name":"amount","type":"uint256"}],"name":"transfer","outputs":[],"payabl
e":false,"stateMutability":"nonpayable","type":"function"},{"constant":true,"
inputs":[{"name":"user","type":"address"}],"name":"getBalance","outputs":[{"n
ame":"","type":"uint256"}],"payable":false,"stateMutability":"view","type":"f
unction"}]
```

Then in the geth terminal, you can associate the abi to the address of the contract:

```
      > var bankAbi =
[{"constant":false,"inputs":[],"name":"withdraw","outputs":[],"payable":false
,"stateMutability":"nonpayable","type":"function"},{"constant":false,"inputs"
:[],"name":"buy","outputs":[],"payable":true,"stateMutability":"payable","typ
e":"function"},{"constant":false,"inputs":[{"name":"to","type":"address"},{"n
ame":"amount","type":"uint256"}],"name":"transfer","outputs":[],"payable":fal
se,"stateMutability":"nonpayable","type":"function"},{"constant":true,"inputs
":[{"name":"user","type":"address"}],"name":"getBalance","outputs":[{"name":"
","type":"uint256"}],"payable":false,"stateMutability":"view","type":"functio
n"}]
      > var MyBank = web3.eth.contract(bankAbi);
      > var MyBankInstance =
MyBank.at('0x38B30b220F0277a179958401CFD638708AA321f2');
```

Now, you can call the functions of the contract by executing a transaction through `MyBankInstance`.

To send a transaction calling buy with 1 ether:

```
      > MyBankInstance.buy({from: eth.accounts[0], value:web3.toWei(1,
'ether')})
      "0x53663a568f65ef3f84ad6b66e260f3f8255188a54d565ad33a86f44b09b15c1b"
```

- The value returned is the hash of the transaction.
- In `{}` you can specify the transaction parameters (`from` specify the sender, `value` the ether sent)
- `eth.accounts` is the list of local accounts
- `web3.toWei` converts ether to wei (1 wei = 1e-18 ether)

You can inspect the transaction using its hash:

```
>eth.getTransactionReceipt('0x53663a568f65ef3f84ad6b66e260f3f8255188a54
d565ad33a86f44b09b15c1b')
```

Or using the ganache GUI.

Some functions do not change the state of the blockchain, and can be executed locally. These functions are marked with one of these attributes:

- constant
- pure
- view

For example, `getBalance` can be called without executing a transaction. To do so, do not specify the transaction parameters `({from: ..})` when you call the function:

```
> MyBankInstance.getBalance(eth.accounts[0])
1000
```

As you see, you now possess 1000 tokens of your contract!

Exploiting the Vulnerability

Another user already bought 90 ether worth of token (see transaction: 0xdd7de39d9e1ab9f098ecaa48e7599f5b4b5fe29ae068cfcdf99b0e457150d6bf)

It is possible to steal the ether?

Let us look more closely at the transfer function:

```
function transfer(address to, uint amount) public{
    tokens[to] += amount;
    tokens[msg.sender] -= amount;
}
```

Figure 3: Buggy transfer function

There is no check to ensure that the sender has enough tokens! If someone sends more tokens than he has, an underflow will occur at:

```
tokens[msg.sender] -= amount;
```

Leading the user to have a large number of tokens!

Once we have an arbitrary amount of tokens, we can withdraw the entire ether balance of the bank by calling withdraw.

To summarize to exploit the vulnerability, you need to:

1. Call transfer with a large number to trigger the underflow
2. Set your token balance to the bank ether balance. You can send tokens to arbitrary addresses to reduce your balance.
3. Call withdraw

1 and 2 can be done in one action if you determine the underflow result.
The exact ether balance of the bank is returned by:

```
> eth.getBalance(MyBankInstance.address)
```

Exploit the contract and steal the ether.

Web3 tips

- You can store the return of a call to a variable:
  - `var my_balance = eth.getBalance(MyBankInstance.address)`
- Due to the internal representation for [large number](#) in web 3, do not use direct arithmetic operations, but use the BigNumber operators:
  - `var_a.plus(var_b)` instead of `a + b`
  - `var_a.sub(var_b)` instead of `a - b`

Exercise 2: Stealing the Ownership

The following contract is deployed at address 0x06aC60dd55774579608fF22740b72042C165526b.
90 ethers are inside the contract, can you steal them?

```
contract IamTheOwner{
  address owner;

  // Constructor
  // In Solidity a constructor can only be called
  // One time.
  // To be a constructor, a function needs to have
```

```
    // the same name as the contract, or
    // to be declared with the constructor keyword
    function IAmTheOwner() public{
        owner = msg.sender;
    }

    // Empty function to deposit ether
    function deposit() payable public{}

    // Withdraw the account balance
    function withdraw() public {
        // It can only be called by the owner
        require(msg.sender == owner);
        owner.transfer(address(this).balance);
    }
}
```

Figure 4: IamTheOwner.sol

Hint

Do not read the following if you don't want help

A constructor is a specific function in Solidity that is only callable when the contract is deployed. To be the constructor a function must:

- Have the *exact* same name as the name
- Use the constructor keyword

The constructor keyword was recently added to Solidity, as several contracts were hacked because a function was supposed to be the constructor, but was incorrectly named, due to typo or change in the contract name.

Exercise 3: Guessing the Number

The following contract is deployed at 0x397B55bB094da054E85AEa66835355D2AcA14645:

```
contract Bet {
    address owner = msg.sender;
    uint random_number = 0x0;

    function setRandomNumber(uint n) payable public{
        require(msg.sender == owner);
        random_number = n;
    }

    function guessNumber(uint guess) public {
        if(random_number == guess){
            owner = msg.sender;
            msg.sender.transfer(address(this).balance);
        }
    }
}
```

Figure 5: Bet.sol

Can you guess the current number currently stored in `random_number`?

Hint

Do not read the following if you don't want help

On ethereum, all transactions are public. As a result, you are able to track what transactions were done to this contract. In particular, there were two transactions:

- 0xdb64c6aa544f1ab48e6aa96f26e2c2d8d9cf4f953893f79f9c1b605c4136a526
- 0x554d16b90e66a5c7661c2df054df44ab9fb8bb573aef2291cbc0c1c04612a638

A transaction has a data field:

- The first four bytes are the function id, which is equal to the keccak hash of the function signature. In our example:
    - 0xd6bfea28: `setRandomNumber(uint256)`
    - 0xb438d018: `guessNumber(uint256)`
- The remaining bytes are the parameters

As a result, you can follow the transactions and discover what was the last number set through `setRandomNumber`.

Exercise 4: Exploiting a Reentrancy

The following contract is deployed at 0x72f4572204D8E5cE165968b1B49AE360a0A50dD3:

```
contract Reentrancy {
   mapping (address => uint) userBalance;

   function getBalance(address u) view public returns(uint){
       return userBalance[u];
   }


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


   function withdrawBalance() public {
       if( ! (msg.sender.call.value(userBalance[msg.sender])() ) ){
           revert();
       }
       userBalance[msg.sender] = 0;
   }
}
```

Figure 6: Reentrancy.sol

2 ethers were deposed, could you steal them?

Hint

Do not read the following if you don't want help

A reentrancy vulnerability is a specific vulnerability that abuses the ethereum design.

A contract can send a transaction to another contract or a wallet.

If the destination is a contract, it’s so-called fallback function will be executed (if present). The fallback function of a contract has the signature:

```
        function () { ...}
```

A malicious fallback function can be able to execute a function of the original contract (and thus re-enter to the contract). On our example, it creates the following attack vector:

- The malicious contract call the `withdrawBalance` function
- The execution of `withdrawBalance` leads to execute the fallback function of the malicious contract
- The fallback function calls a second times `withdrawBalance`
- As a result, the malicious contract can withdraw two times its original balance

We will use the following skeleton of malicious contract:

```
// ABI of the target
contract Reentrancy {
   function addToBalance() payable public;
   function withdrawBalance() public;
}

contract ReentrancyExploit {
  bool attack_is_on=false; // set to true when the attack is on
  Reentrancy vulnerable_contract;
  address owner;

  constructor() public{
      owner = msg.sender;
  }

  // Depose some ether to the target
  // And save the target address
  function deposit(Reentrancy _vulnerable_contract) public payable{
      vulnerable_contract = _vulnerable_contract ;
      vulnerable_contract.addToBalance.value(msg.value)();
  }
  // This function will launch the attack
  function launch_attack() public{
      // You must set attack_is_on to true
      // and call the withdrawBalance function of the target
  }

  // fallback function
  // This is the function executed when a transaction is sent to
  // the contract
  function () public payable{
      // You must check that the attack is ongoing
      // If it is, re-call the withdrawBalance of the target
      // and set attack_is_on to false to prevent an infinite loop
  }

  // Withdraw the money
  // This function will be call after the attack, to get the money
  function get_money() public{
      owner.transfer(addressthis).balance);
  }
}
```

Figure 7: ReentrancyExploit.sol

You will need to update:

- The `launch_attack` function
- The fallback function

Once you have updated the contract, you can deploy it, using truffle:

```
$ truffle migrate -f 3
```

The address of the contract will be display:

```
ReentrancyExploit: 0x..
```

Update ReentrancyExploit.sol and use it to exploit Reentrancy.sol.