Design Patterns and Security

By Code Eater

Contents

Fallback Function

- The fallback function is executed on a call to the contract if none of the other functions match the given function signature, or if no data was supplied at all and there is no receive Ether function.
- Fallback function is optionally payable.
- A contract can have at most one fallback function, declared using either fallback () external [payable] or fallback (bytes calldata input) external [payable] returns (bytes memory output)
- This function must have external visibility.
- A fallback function can be virtual, can override and can have modifiers.
- The fallback function always receives data, but in order to also receive Ether it must be marked payable.

Receive

- A contract can have at most one receive function, declared using receive() external payable { ... }.
- This function cannot have arguments, cannot return anything and must have external visibility and payable state mutability.
- It can be virtual, can override and can have modifiers.
- If neither a receive Ether nor a payable fallback function is present, the contract cannot receive Ether through regular transactions and throws an exception.
- Using payable fallback functions for receiving Ether is not recommended.

Self-Destruct

- The remaining Ether stored at that address is sent to a designated target and then the storage and code is removed from the state.

- Removing the contract in theory sounds like a good idea, but it is potentially dangerous, as if someone sends Ether to removed contracts, the Ether is forever lost.

- Even if a contract is removed by selfdestruct, it is still part of the history of the blockchain and probably retained by most Ethereum nodes. So using selfdestruct is not the same as deleting data from a hard disk.

Dangerous Code

```
contract demo {
address public manager;

constructor(){
    manager=msg.sender;
}
function sendEther(address receiver) public payable {
    payable(receiver).transfer(msg.value);
}

function sendContractEth() public payable{

}
function contractBalance() public view returns(uint){
    return address(this).balance;
}
function destroy() public {
    require(manager==msg.sender,"You are not the manager");
    selfdestruct(payable(manager));
}
}
```

Self-Destruct Pattern

```
contract demo {
address public manager;
constructor(){
    manager=msg.sender;
}
bool public destoryed;
modifier isNotdestroyed(){
    require(destoryed!=true,"Contract is destroyed");
    _;
}
function sendEther(address receiver) public payable isNotdestroyed{
    payable(receiver).transfer(msg.value);
}

function sendContractEth() public payable isNotdestroyed {
}
function contractBalance() public view isNotdestroyed returns(uint)  {
    return address(this).balance;
}
function destroy() public isNotdestroyed{
    require(msg.sender==manager,"You are not the manager");
    payable(manager).transfer(address(this).balance);
    destoryed=true;
}
receive() external payable isNotdestroyed{

}

}
```

Factory Pattern

```
contract demo {
 Computer[] public arr;
 function instance() public {
   arr.push(new Computer());
 }
}

contract Computer{

}
```

Mapping Iterator

```
contract demo {
 mapping(address=>uint) public tokens;
 address[] public arr;
 function tranfer(uint amount) public {
   tokens[msg.sender]=amount;
   arr.push(msg.sender);
 }

}
```

Name Registry

```
contract Company {
 mapping(string=>address) public store;
 function storeAddress() public {
   store["CEO"]=address(new CEO());
   store["HR"]=address(new HR());
   store["Manager"]=address(new Manager());
 }

}

contract CEO {

}

contract HR {

}

contract Manager {

}
```

Withdrawal Pattern

- Design a contract to find out the richest person.
- Every person in this contract will be able to donate ether.
- From these donated ether find out the richest person.
- Once you have the richest person return the ether donated by the address back to that address.
- Store the richest person address and the amount that person donated.

Withdrawal Pattern

```
contract withdraw{
    address richest;
    uint max;

    constructor() payable{
        richest=payable(msg.sender);
        max=msg.value;
        payable(richest).transfer(msg.value);
    }
    function sendEther() payable public{
        require(msg.value>max,"You are not the richest");
        richest=payable(msg.sender);
        max=msg.value;
        payable(richest).transfer(msg.value);
    }
}

contract senderContract{
    //sendEther() call
}
```

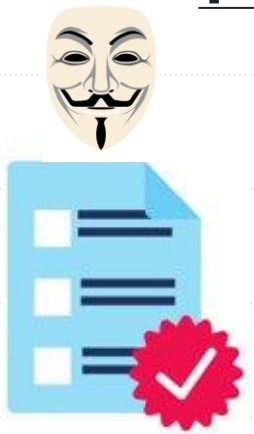
Withdrawal Pattern

```
contract withdrawPattern{
    address richest;
    uint max;
    mapping(address=>uint) investors;//new
    constructor() payable{
        richest=payable(msg.sender);
        max=msg.value;
        payable(richest).transfer(msg.value);
    }
    function sendEther() payable public{
        require(msg.value>max,"You are not the richest");
        richest=payable(msg.sender);
        max=msg.value;
        //payable(richest).transfer(msg.value);
        investors[msg.sender]=msg.value; //new
    }

    function withdraw() public {
        uint amount=investors[msg.sender];
        investors[msg.sender]=0;
        payable(msg.sender).transfer(amount);
    }
}
```

Re-entrancy Attack

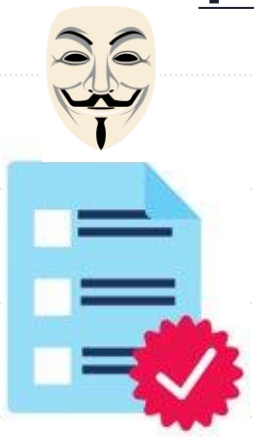
Re-entrancy Attack



```
receive external payable{
requestEther();
}
```

```
function requestEther(){
address.transfer(balance);
balance[address]=0;
}
```

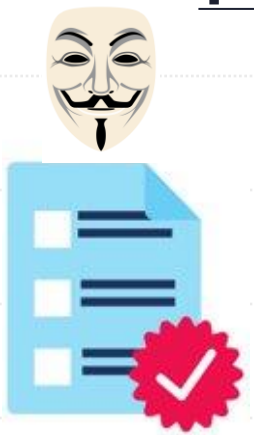
Re-entrancy Attack



```
receive external payable{
requestEther();
}
```

```
function requestEther(){
address.transfer(balance);
balance[address]=0;
}
```

Re-entrancy Attack



```
receive external payable{
requestEther();
}
```

```
function requestEther(){
uint temp=balance[address];
balance[address]=0;
address.transfer(temp);
}
```