1. Reentrancy Attacks

Reentrancy attacks

In this assignment we'll create a *reentrancy attack* on a vulnerable smart contract.

Consensys has a good guide on reentrancy attacks.

There have been many reentrancy attacks in the real world.
The DAO Attack is the largest and most well-known example of a reentrancy attack, but there have been many others.

Assignment

In this assignment, you'll execute a reentrancy attack on a vulnerable contract.

We've prepared a contract "Bank.sol" which is vulnerable to a reentrancy attack through an ERC777 token withdrawal.

The Bank contract works a bit like wETH. Users can deposit ETH into the Bank contract, and in return they receive a new ERC777 token. At a later point, they can redeem the ERC777 token for the underlying ETH held in the contract.
The process goes like this:

1. Users call the "deposit()" function and send ETH to the contract. This increments the user's local balance inside the Bank contract.
2. Users who have a positive balance can "withdraw" ERC777 tokens by calling the claimAll() functions (this is the function that is vulnerable to a reentrancy attack).
3. Users can redeem their ERC777 tokens for ETH by calling the redeem() function

If you were to build a contract like this in the real world, you would probably combine Steps 1 & 2, but that would eliminate the claimAll() function, and there is no simple opportunity for reentrancy in the deposit() function (since you must provide ETH with each call), thus this assignment would have to have a reentrancy vulnerability in the redeem() function.
In that case, you would be stealing ETH from the contract (instead of stealing ERC777 tokens). This would mean that you can only steal as much ETH as the contract has, and your attacker
code would be more complicated, since you would have to calculate the amount you could steal before executing the reentrancy attack. As it stands now, you are stealing ERC777 tokens
from the contract, but these are minted on demand by the contract, so the contract can never run out.

Your assignment is to complete "Attacker.sol".

1. setTarget - This tells the attacker contract the address of the target Bank contract.
2. attack - This executes the attack by calling the deposit() function and then the vulnerable withdraw function on the Bank contract (look in Bank.sol to identify this function).
3. withdraw - This function allows the owner of Attacker contract to withdraw the stolen ERC777 tokens to a given account.
4. tokensReceived - This function is required by the ERC777 token standard, and it's what allows the reentrancy attack.

You will need to complete the functions “attack” and “tokensReceived” – the others have been completed for you.

Testing with Foundry

This assignment uses the https://book.getfoundry.sh/ testing environment. You can use foundry locally to test your solution, and our autograder uses Foundry as well.

You are not required to use Foundry in any way, but it will probably help in debugging your code.

The autograder will perform the following steps:

1. deploy a vulnerable “Bank” contract
2. deploy your “Attack” contract
3. call “setTarget” on the Attacker contract to inform the Attacker about the vulnerable Bank contract
4. call “attack” on your Attacker contract

More information about reentrancy and ERC777 tokens

ETH is vulnerable to reentrancy

Below is the canonical example of a deposit contract that is vulnerable to a reentrancy attack

```
contract DepositFunds { //Vulnerable contract
    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);

        //The following line transfers bal ETH to msg.sender
```

```
        //If msg.sender is a contract, that contract takes over control fl
        //A malicious contract can call withdraw() again here
        msg.sender.call{value: bal}("");

        balances[msg.sender] = 0;
    }
}
```

Reentrancy with ERC20 tokens

The reentrancy vulnerability comes about because transferring ETH *calls* the recipient, and if the recipient is a contract, then the recipient can hijack control flow at this point.
The reentrancy problem would seem to go away with ERC20 tokens

```
contract DepositFunds { //Possibly vulnerable contract
    mapping(address => uint) public balances;
    ERC20 token;

    function deposit(uint256 amount) public {
        token.transferFrom(msg.sender,address(this),amount);
        balances[msg.sender] += amount;
    }

    function withdraw() public {
        uint bal = balances[msg.sender];
        require(bal > 0);

        //In *most* ERC20 tokens, calling the transfer function does *n
        //If the transfer function *does* call the recipient, then this
        //So this contract is vulnerable for some tokens, but not for o
        token.transfer(msg.sender,bal);

        balances[msg.sender] = 0;
    }
}
```

The ERC777 token standard extends the ERC20 token standard, and one of the extensions is the introduction of receive hooks, which means that ERC777-compliant tokens *should* call the recipient.

ERC777

The ERC777 token standard is an extension of the ERC20 standard that has a few new features. In this assignment, we'll only be using two features:

1. ERC777TokensRecipient - With an ERC20 token (or ETH), if you mistakenly send tokens to an Externally Owned Account (EOA), you can, potentially, ask the account owner to return the tokens. If you mistakenly send tokens to a *contract* the contract can only return the tokens if it was programmed to do so (which is unlikely). Apparently, it is common for users to send their ERC20 tokens back to the contract that controls them. You can see this behavior on Etherscan. For example, people have sent over 300,000 USDT to the USDT contract. USDT holders seem particularly confused, as they've also sent 47,000 USDT to the USDC contract. To fix this problem, the ERC777 standard specifies the notion of an ERC777TokensRecipient. To mitigate this problem, the OpenZeppelin ERC777 contract looks in the ERC1820 registry to see if the intended recipient has registered.
2. Receive hooks - When an ERC777 token is transferred, the contract calls the TokensReceived function on the recipient contract. This is what allows for reentrancy attacks. Your contract should do two things
    1. Register with the EIP1820 Registry to be able to receive ERC777 tokens
    2. Complete the tokensReceived function – this will be the function that executes the recursive withdrawal

For example, the OpenZeppelin's ERC777 implementation calls the recipient on a transfer.

For more information on ERC777 tokens, see the ERC777 tutorial on EatTheBlocks.

The Uniswap attacks

Uniswap v1 was vulnerable to reentrancy attack. OpenZeppelin has a nice guide illustrating the attack.

The vulnerability was removed in Uniswap v2 through the addition of the "lock modifier". When you enter basically any of the functions on the Uniswap Pair contract, the "unlocked" variable is set to 0, and no one else is allowed to call the function until the "unlocked" variable is reset to 1.

Delicacy of reentrancy attacks

Reentrancy attacks are actually quite delicate. Even slight changes to the code can remove the vulnerability.

```
contract DepositFunds { //NOT Vulnerable
    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);

        balances[msg.sender] = 0; //Moving this line above the transfer

        msg.sender.call{value: bal}(""); //If attacker calls withdraw()

    }
}
```

```
contract DepositFunds { //NOT Vulnerable
    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);

        msg.sender.call{value: bal}(""); //Attacker gets control flow h

        balances[msg.sender] -= bal; //If attacker made a recursive cal

    }
}
```

Next