“Analysis of Reentrancy Attack Vulnerabilities in Smart Contracts”

Github repo: https://github.com/roshannp/Re-entrancy

Roshan Nellore Prasad - rone7552@colorado.edu

Nikhil Marri - nima6198@colorado.edu

Table of Contents

Abstract:

Our research project focused on investigating the reentrancy attack vulnerability in smart contracts. Our basic goal was to develop a comprehensive understanding of the mechanisms behind this type of attack. To achieve this, we analyzed various past instances of reentrancy attacks and experimented with implementing a basic version of the attack in a smart contract using the Remix progressive environment. It also examines how the attack operates, we also conducted a detailed analysis of the security measures that can be employed to prevent or mitigate its effects. Our research project aimed to provide a thorough and comprehensive understanding of the reentrancy attack vulnerability in smart contracts and its associated security issues.

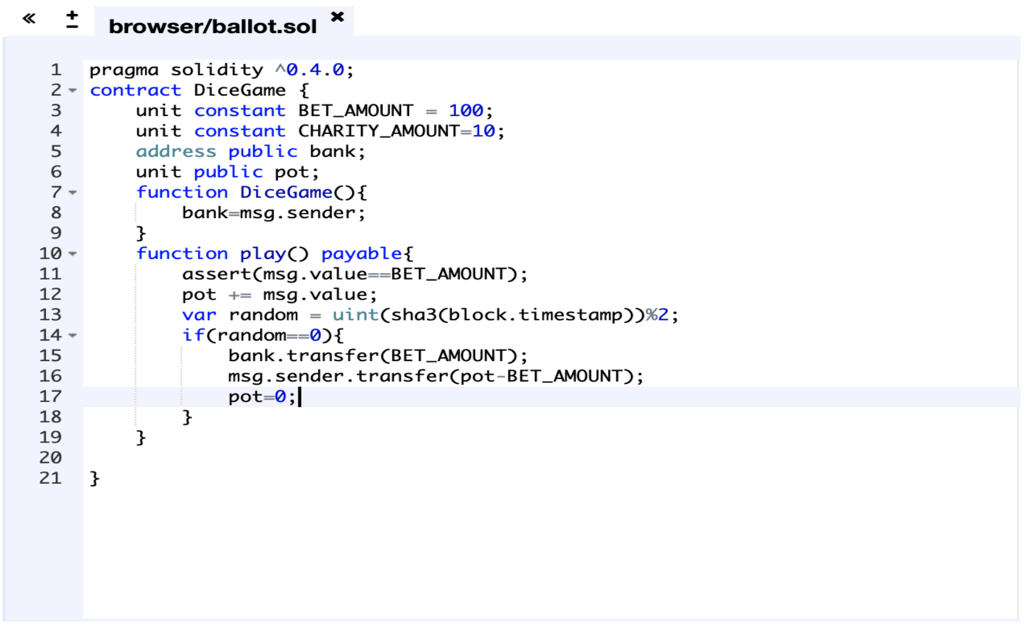
Introduction:

In blockchain terminology, A smart contract can be defined as the digital agreement of a computer program that is executed when certain conditions are met. It is mostly used to optimize the processes between unknown people without the necessity of a middleman. Smart contracts have traits like transparency, security, and efficiency. It is very remarkable to have immutability in blockchain technology. Smart contracts use secure and transparent technology. Its immutability makes it difficult to alter transactions, and its decentralization ensures that there is no single point of failure. Smart contracts need to be very accurate if any part of the program is not well written there are high chances of getting exploited in the public domain. One such attack is called the "Reentrancy attack."Reentrancy is one of the vital problems in a smart contract. This vulnerability occurs when a smart contract makes an external call to another contract. When an external call is made the execution of EVM is passed from the initial contract to the other. The external call is not safe because contract B can do anything it wants. It could call another contract or execute certain malicious logic inside of its own scope. Contract B can call Contract A again. Generally, Reentrancy occurs when the contract is not up to the mark, where the function makes an external call to a malicious contract. Generally, the malicious contract is created by threat actors to loot Ethereum from the victim's account. The main reason for this attack is that anyone could investigate smart contracts because of the transparency of blockchain technology. The key steps we can take to reduce reentrancy is to check the code multiple times and use secure coding practices"[1]. Also, the other common way is to limit the amount of gas for every function call.

Background:

Reentrancy attack occurs in multiple forms Timestamp dependency attack, Front running attack, Cross Functional call attack, and Recursive call attack.

a) **Timestamp dependency attack:** Blockchain networks are distributed, so it's hard to set and sync the exact time between nodes. This is a challenge for smart contracts, which need to know the current time. When a smart contract relies on the block timestamp to perform an operation, it can be vulnerable to attack because the timestamp can be controlled by miners.



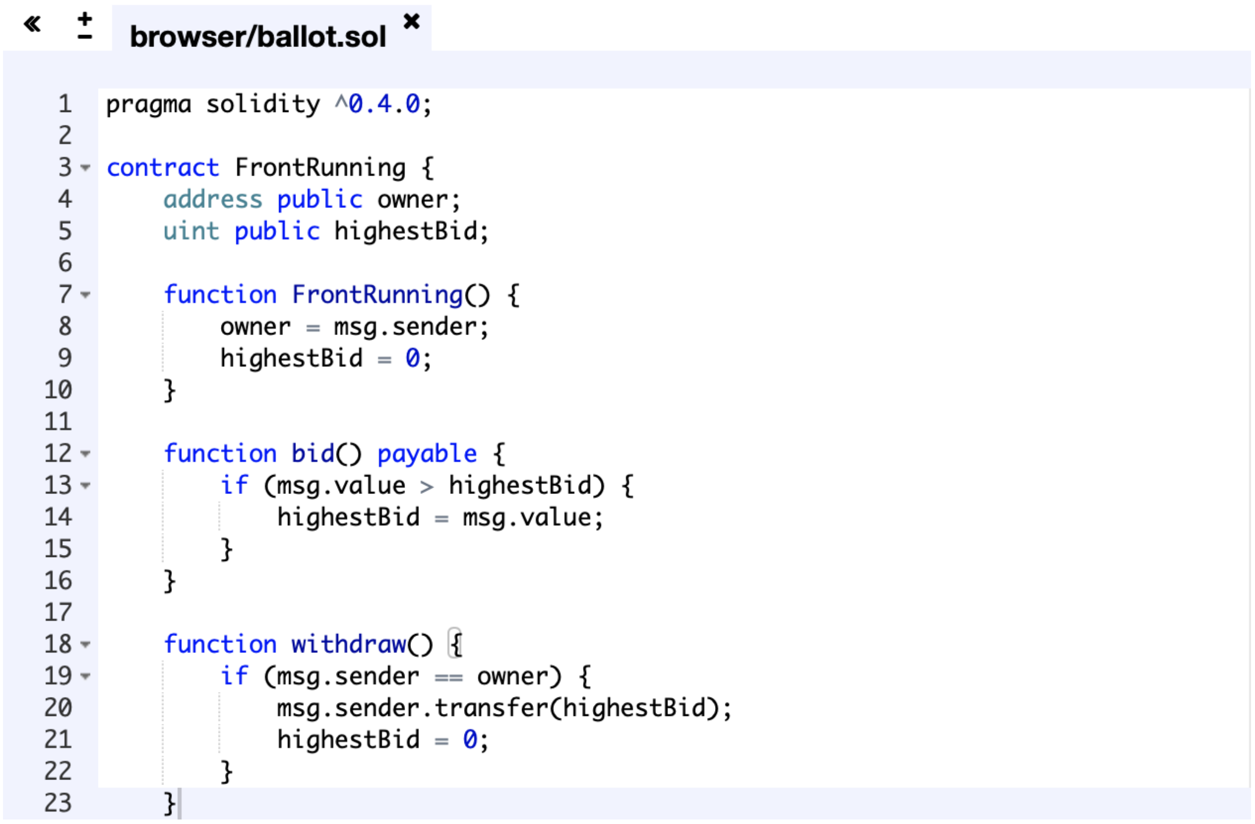
```
pragma solidity ^0.4.0;
contract DiceGame {
    unit constant BET_AMOUNT = 100;
    unit constant CHARITY_AMOUNT=10;
    address public bank;
    unit public pot;
    function DiceGame(){
        bank=msg.sender;
    }
    function play() payable{
        assert(msg.value==BET_AMOUNT);
        pot += msg.value;
        var random = uint(sha3(block.timestamp))%2;
        if(random==0){
            bank.transfer(BET_AMOUNT);
            msg.sender.transfer(pot-BET_AMOUNT);
            pot=0;
        }
    }

}
```

The code has a timestamp reentrancy attack because it uses the block timestamp to generate a random number. “The block timestamp can be manipulated by an attacker, which allows them to repeatedly call the play() function and exploit the random number generation.

An attacker can do this by calling the play() function multiple times within the same block, each time setting the timestamp to a slightly earlier time than the previous call. This would cause the same random number to be generated each time, allowing the attacker to predict the outcome of the game and exploit it for their own gain”[2].

b) Front-running reentrancy attack:

“Front-running is a type of attack where an intruder can see and respond to a transaction before it is included in a block. This is possible because all transactions are noticeable in the MEMPOOL for a brief time before being executed. An example of how this can be exploited is with a decentralized exchange. Say a user wants to buy 1 ETH. The user sends a transaction to the exchange to buy 1 ETH. The transaction is visible in the MEMPOOL so that an attacker can see it. The attacker can then send their own transaction to sell 1 ETH, and their transaction will be executed before the user's transaction. This means that the attacker will be able to buy the ETH at a lower price than the user”[3].

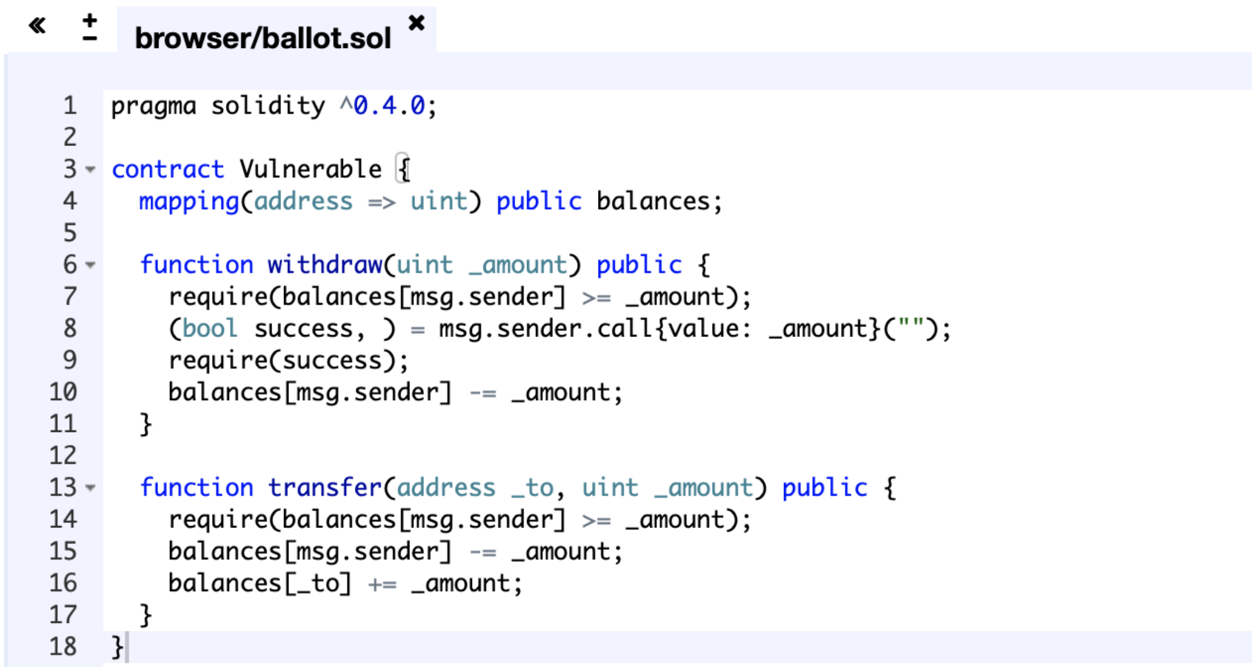


```
pragma solidity ^0.4.0;

contract FrontRunning {
    address public owner;
    uint public highestBid;

    function FrontRunning() {
        owner = msg.sender;
        highestBid = 0;
    }

    function bid() payable {
        if (msg.value > highestBid) {
            highestBid = msg.value;
        }
    }

    function withdraw() {
        if (msg.sender == owner) {
            msg.sender.transfer(highestBid);
            highestBid = 0;
        }
    }
```

This smart contract allows users to bid and withdraw their bids. It does, however, have a vulnerability to front-running attacks. This occurs because the bid() method does not confirm the sequence of transactions. An attacker may observe a large bid transaction and rapidly put in a greater bid before the first one is completed. This causes the first bid to fail and the attacker's bid to succeed. To circumvent front-running attacks, smart contracts should be designed to rely less on transaction sequence and instead employ strategies such as commit-reveal schemes. This makes it more difficult for attackers to snoop on transactions before they are finished.

c) Cross-function reentrancy attack:

Cross-function reentrancy occurs when a function that was not initially called is reentered during an attack, as opposed to traditional reentrancy attacks that call the same function. This form of assault may be more difficult to detect since complex protocols have many alternative outcomes, and it may be impossible to test each potential event separately.



```
pragma solidity ^0.4.0;

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

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

  function transfer(address _to, uint _amount) public {
    require(balances[msg.sender] >= _amount);
    balances[msg.sender] -= _amount;
    balances[_to] += _amount;
  }
}
```

“There are two functions in this contract: withdraw() and transfer(). The withdraw() function is vulnerable to a cross-function reentrancy attack since it utilizes call() to send Ether to a user's address. An attacker can regularly deplete the contract's balance by executing the transfer() method and then calling back into the withdraw() function through a fallback mechanism. To counter this attack, the withdraw() function should send Ether using send() or transfer() rather than call(), limiting the gas accessible to the receiving contract and stopping it from callback into the susceptible function”[4].

d) Recursive call reentrancy attack:

The attacker calls a vulnerable contract continuously, which might result in the execution of an undesired function or the execution of the same function several times.

History of Reentrancy Attack:

The most notorious Ethereum exploit Dao Hack is the justification for the ascent of inquiries on the Reentrancy assault in 2016. The burger swap hack, BNB hack, and cream finance hack had the reentrancy exploit part.

a) DAO Attack:

A Decentralized Autonomous Organization (DAO) is a set of collections of guidelines given by the Blockchain-dependent cooperative team. It is one of the shocking events that happened in blockchain history in 2016. The DAO was hacked after raising funds of one hundred and fifty million dollars in the Ethereum token sale because of the susceptibility present in the base code. Before the conclusion of the token selling, clients raised the base code issues. When the software engineers tried to solve the issue in the meanwhile one of the attackers Exploited the Issue and tried to empty out the funds in the DAO. The response to this hack is to propose a soft fork from Ethereum stating that by including a few lines of code snippets that help to block the attacker which in turn helps in preventing this kind of attack. "However, the attacker issued a paper document to the Ethereum community in the Slack group channel. The letterhead says the assets are asserted by the assailant legally. The Ethereum community gave a solution called Hard Fork which means the blockchains should be immutable. The hard fork is executed later with many debates to proceed to result in the separation of the Ethereum community into two"[5].

b) Burger Swap Hack:

Burger Swap Hack is basically a fake token exchange that happened with a flash loan attack. This attack is done due to continuous changes in burger token value. "It was a flash loan from Pancake Swap in providing the sources of funds. The attacker has created a fake token and performed a burger swap by done trading from flash loan to burger on burger swap. The attacker also added a significant amount of fake tokens and convert those fake tokens to WBNB using the pool and swap those WBNB for some Burger Tokens. With the help of this, the attacker steals the total amount of 7.2 million dollars"[6]. This incident made the analysts decide that it depends on exchange rates and try to prevent the usage of exchange rates by smart contracts.

c) BNB Hack:

BNB was the highest trade exchange where the security threat takes place accidentally due to an internal error that gives the loss of two BNB tokens in millions (2 million BNB tokens) which means five hundred and seventy million dollars (570 million $) in money to the company. The threat is done with respect to the cross-chain bridge. It involves shifting crypto assets from the crypto blockchain of one to another. "The hacker is actually able to damage the Binance bridge and forward a single BNB token in million for two times. This attack is done due to an error in the smart contract transactions. This attack basically spoils the exchange and forwards the necessary beneficial resources to the respective crypto wallets". This is an example of one of the risks of joining digital assets for newcomers.

d) Cream-Finance Hack:

It is similar to the burger swap hack since it is a flash loan attack. It is also recorded as the largest theft with a worth of one hundred and thirty million dollars(130 million $). The attacker utilized a flash loan attack that exploited a severely executed Oracle price proxy. This permitted the assailant to control the cost and then utilize yUSD to acquire from many business sectors". To prevent this kind of attack Utilize a legitimate oracle to get the cost. Another great measure is adding an asset limit for each pool in USD that a particular party can do inside a specific time period. This would allow outsiders to have a chance to respond in the event that something turns out badly.

e) Fei Protocol Hack:

It is one of the reentrancy attacks that result in the loss of almost Eighty million dollars(80 million $) in the form of digital assets like tokens. On account of the Fei Convention, its utilization of code forked from the Compound set it in danger. Inside the CEther code, numerous reentrancy weaknesses were fixed in a past update, however, a few weak capabilities were disregarded. "This attack happened because of two main components and they are the exit market and the borrow. The assailant took advantage of this weakness by calling get utilizing a brilliant agreement address. When the borrowing capability sends the credited sum to the borrower, it has not yet refreshed its inside state to mirror the way that the stored resource is right now being utilized as security. The exit market capability checks that a store is not generally utilized as a guarantee for credit and afterward permits it to be removed. The borrow capability permits a client to apply for a line of credit involving a stored resource as security and doesn't follow the check-impact communication design, leaving it helpless against assault. On the off chance that the Fei Convention had gone through a security review, almost certainly, this weakness would have been found and fixed before it cost the convention and its clients $80 million.

f) Parity Wallet Hack:

The parity wallet hack is the second biggest hack, as far as ETH stealing. The assailant sent two exchanges to every one of the impacted agreements: "the first to get selective responsibility for MultiSig and the second to move its assets. The assailant was then the main proprietor of the multi-sig, actually depleting the agreement of every one of its assets. "The attack might have been stopped either by not extricating the constructor rationale into the base code". In the base code, the INITWALLET function is called when constructing a new wallet to set up its initial state. There is also one more function the kill function which is essentially a call to self-destruct. Every multi-signature parity wallet deployed by the users since 2017"[7]. To make it more secure one thing is not to provide any harmful functionality in library Contracts. If the kill function has not been provided in the library contract then the intruder might not be able to do anything to steal the resources from the company.

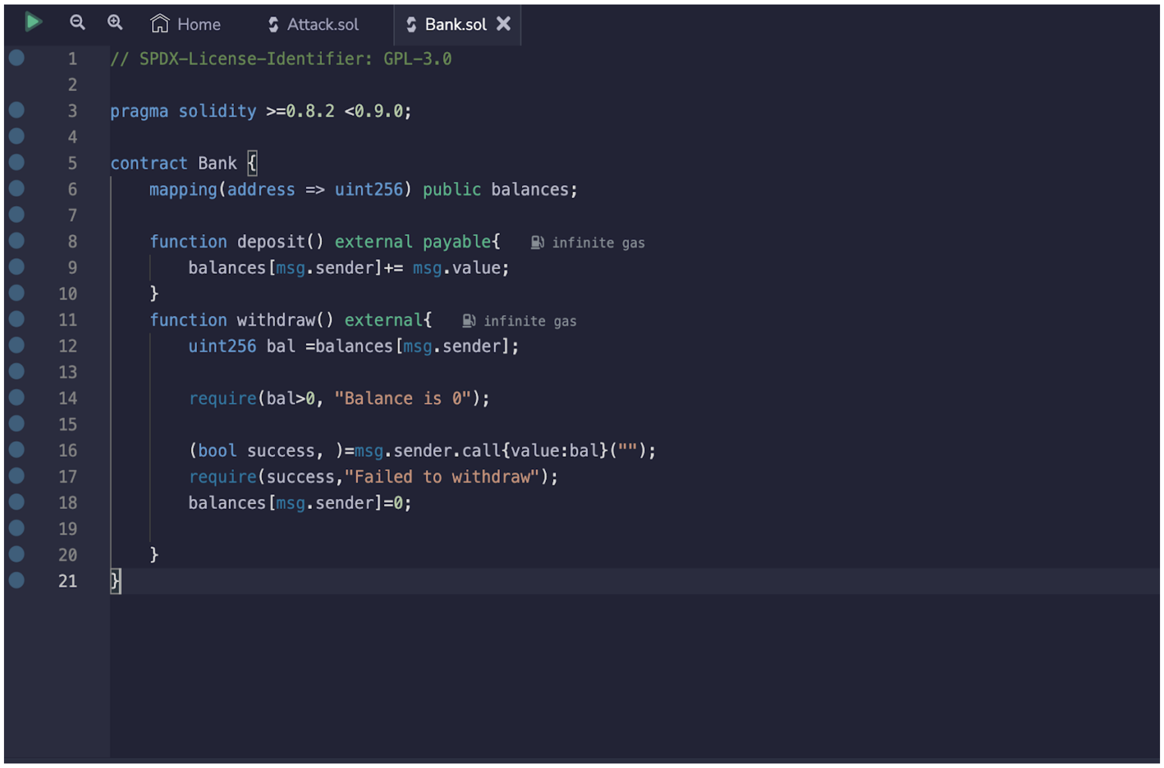
g) NFT lending platform OMNI:

OMNI was hacked with a Reentrancy attack and $1.43 million USD worth of ETH was stolen. For the attack, the attacker initially set up Doodles as security for advances from Wrapped Ethereum (WETH). “The excess Doodle balance on the Omni NFT loaning stage wasn't sufficient to take care of the obligation. For this situation, the framework shuts the position and gives the programmer any Doodles that are still left finished. The Doodles that were purchased with the principal advance were utilized as a guarantee to get more wrapped Ethereum”.Since the stage couldn't see that there was a subsequent obligation, it was simple for the programmer to get more NFTs without taking care of the main credit.

Creating Smart Contract with Vulnerability:

We have created two smart contracts; a) Bank Smart Contract and b) Attacker Smart Contract. The Bank Smart Contract has a vulnerability that is prone to reentrancy attack, the Intruder takes advantage of this by calling from Attack smart contract. We have shown how Vulnerability is exploited, and secured with few paradigms.

a) Bank Smart Contract:



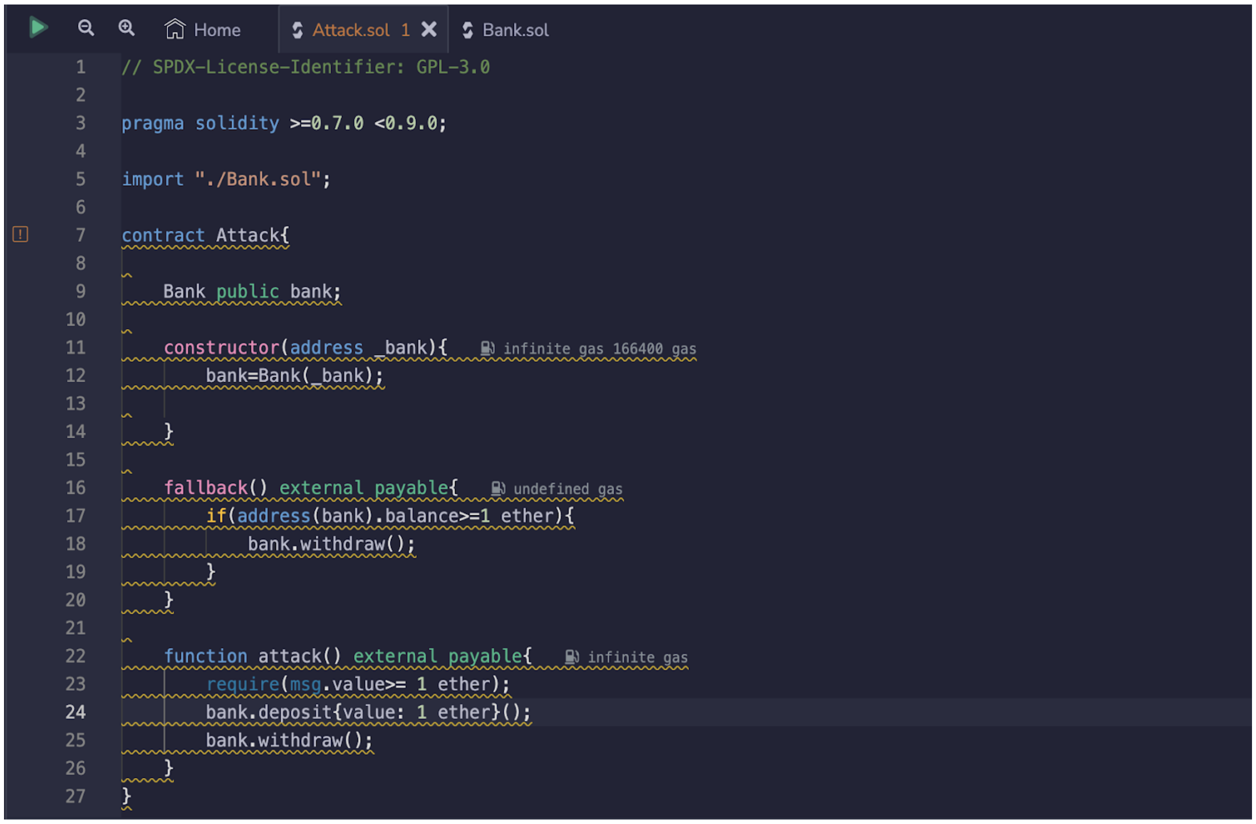
```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.8.2 <0.9.0;

contract Bank {
    mapping(address => uint256) public balances;

    function deposit() external payable{    infinite gas
        balances[msg.sender]+= msg.value;
    }
    function withdraw() external{    infinite gas
        uint256 bal =balances[msg.sender];

        require(bal>0, "Balance is 0");

        (bool success, )=msg.sender.call{value:bal}("");
        require(success,"Failed to withdraw");
        balances[msg.sender]=0;

    }
}
```

“This code defines a basic smart contract called Bank, which allows customers to deposit and withdraw Ether. The balance mapping contains the balance of each user's account. The deposit function lets users increment their accounts by depositing Ether to the contract. Users can withdraw their funds from the contract using the withdraw option.

Before initiating the transaction to send funds to the user's account, the withdraw function first ensures that the user has some amount of balance. The function executes only if the balance is greater than zero. If the balance is greater than zero, the call function is used to transfer Ether from the contract to the user's account. The balance mapping is then updated to reflect the user's withdrawal of funds”.

.

b) Attacker Smart Contract:



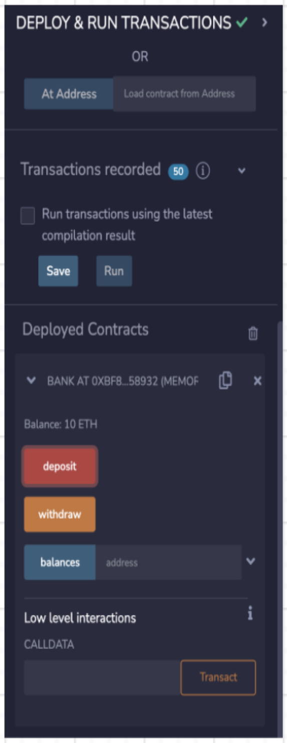
```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.7.0 <0.9.0;

import "./Bank.sol";

contract Attack{

    Bank public bank;

    constructor(address _bank){
        bank=Bank(_bank);

    }

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

    function attack() external payable{
        require(msg.value>= 1 ether);
        bank.deposit{value: 1 ether}();
        bank.withdraw();
    }
}
```

The code defines the Attack contract, which interacts with the Bank contract. In the constructor of the Attack contract, an instance of the Bank contract is created by passing its address as an argument. The instance is stored in a public variable named bank. In Solidity, the fallback function is a particular function that is called when the contract gets a transaction with no function identifier or Ether without invoking any function.

In this scenario, the fallback function checks to see if the Bank contract's balance is more than or equal to one ether, and if so, it invokes the Bank contract's withdrawal function. The attack function is another externally callable function of the Attack contract. The attacker has to contribute at least one ether to this function before invoking it. The deposit function of the Bank contract is called with a value of 1 ether within the attack function. The Bank contract's withdrawal function is then invoked.

Exploiting the Smart Contract:

Bank with Balance of 10 Eth

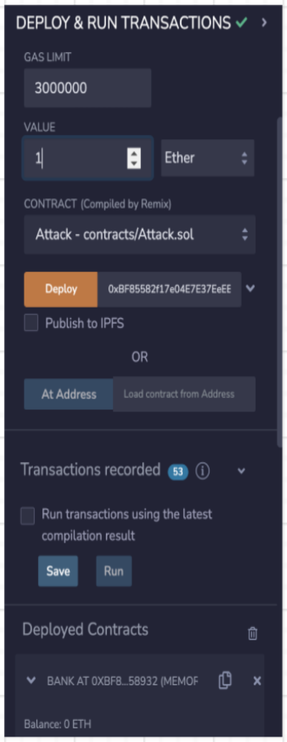


BANK AT 0XBF8...58932 (MEMOF

Balance: 10 ETH

Fig 1.1

Hacker starting attack with 1 Eth


DEPLOY & RUN TRANSACTIONS
Deploy
Publish to IPFS
OR
At Address
Load contract from Address
Transactions recorded 53
Run transactions using the latest compilation result
Save
Run
Deployed Contracts

GAS LIMIT 3000000

VALUE 1 Ether

CONTRACT (Compiled by Remix) Attack - contracts/Attack.sol

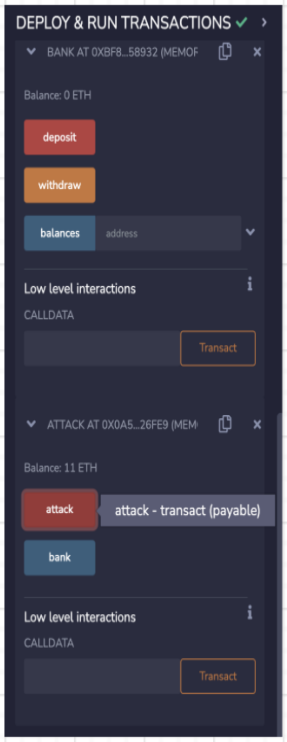
0xBFB5582f17e04E7E37EeEE

BANK AT 0XBF8...58932 (MEMOF

Balance: 0 ETH

Fig 1.2

Attacker with all the balance



BANK AT 0XBF8...58932 (MEMOF

Balance: 0 ETH

ATTACK AT 0X0A5...26FE9 (MEM

Balance: 11 ETH

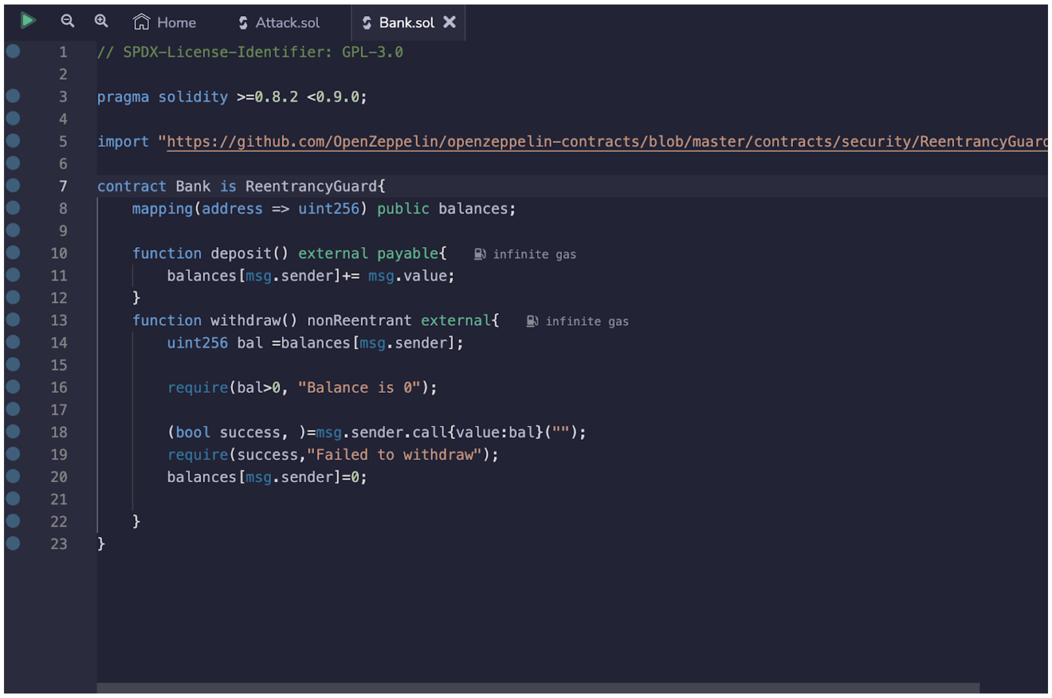
Fig 1.3

Fig 1.1 represents the Bank account with 10 Etherium deposited in it. As the Balance is greater than 0 and has a vulnerability in the contract. Malicious users try to take advantage of the vulnerable function, The **attack** function initiates the attack (Fig 1.2). It first checks that the transaction contains at least 1 ether. Then, it calls the **deposit** function of the bank contract, sending 1 ether with it. Finally, it calls the **withdraw** function of the **bank** contract to withdraw the 1 ether deposited in the previous step. Finally, all the funds are transferred to the Attackers account(Fig 1.3)

Securing the Smart Contract:

a) Reentrancy Gaurd:

“Reentrancy guard is a pattern used in smart contracts to prevent reentrancy attacks. It works by taking advantage of the fact that smart contracts can be called multiple times, either recursively or in succession. This is done by using a state variable to track whether the contract is being called recursively. If the contract is currently being called recursively, the reentrancy guard will prevent the contract from being called again”[10].



```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.8.2 <0.9.0;

import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/security/ReentrancyGuarc

contract Bank is ReentrancyGuard{
    mapping(address => uint256) public balances;

    function deposit() external payable{
        balances[msg.sender]+= msg.value;
    }
    function withdraw() nonReentrant external{
        uint256 bal =balances[msg.sender];

        require(bal>0, "Balance is 0");

        (bool success, )=msg.sender.call{value:bal}("");
        require(success,"Failed to withdraw");
        balances[msg.sender]=0;

    }
}
```

Here, Bank Contract uses the ReentrancyGuard from the OpenZeppelin library to prevent reentrancy attacks. “The ReentrancyGuard contract implements a reentrancy guard by using a boolean flag to check if a function is running. In the bank contract, payment functionality is marked from the non-reentrant modifier inherited from the ReentrancyGuard contract. This modifier ensures that the function is executed only once and prevents reentrant calls”.[10] The non-reentrant modifier sets a boolean flag at the beginning of the function and checks it at the end of the function to make sure no other calls are in progress. By using the ReentrancyGuard contract and marking the withdrawal function with the non-reentrant modifier, this code prevents reentrancy attacks on the bank contract. This is because the modifier ensures that the function will not be re-executed until the previous execution has been completed, and the

ReentrancyGuard contract prevents another function from calling the retired function until the previous call has been completed.

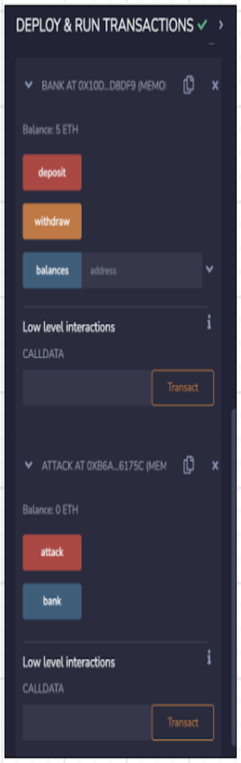
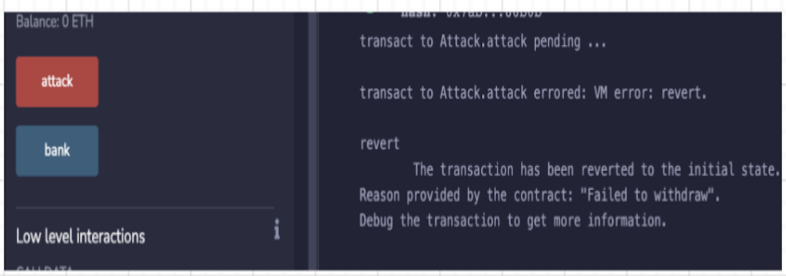


Fig 1.4

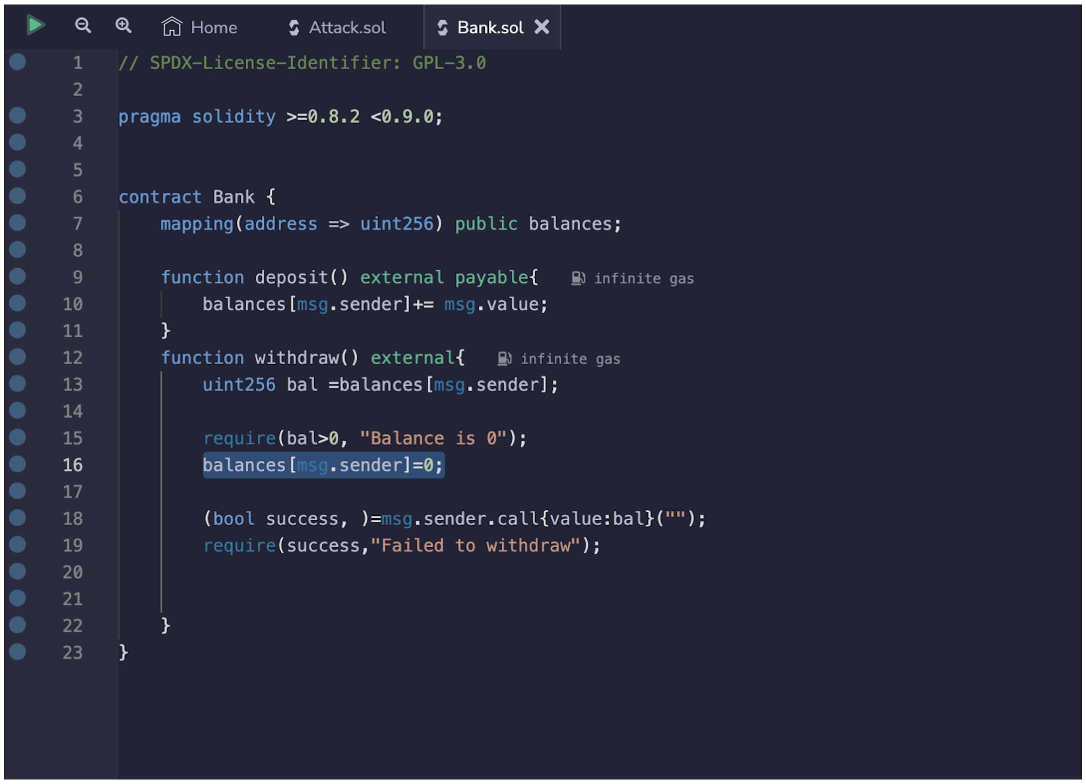


```
transact to Attack.attack pending ...

transact to Attack.attack errored: VM error: revert.

revert
        The transaction has been reverted to the initial state.
Reason provided by the contract: "Failed to withdraw".
Debug the transaction to get more information.
```

Fig 1.5

Here, the Bank has a Balance of 5 Eth (Fig 1.4), when we tried to attack the contract we got an error saying **"The transaction has been reverted to the initial state"** (Fig 1.5)

b) Preventing Reentrancy with State Update:



```solidity
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.8.2 <0.9.0;


contract Bank {
    mapping(address => uint256) public balances;

    function deposit() external payable{    infinite gas
        balances[msg.sender]+= msg.value;
    }
    function withdraw() external{    infinite gas
        uint256 bal =balances[msg.sender];

        require(bal>0, "Balance is 0");
        balances[msg.sender]=0;

        (bool success, )=msg.sender.call{value:bal}("");
        require(success,"Failed to withdraw");


    }
}
```

“The above contract is not prone to a reentrancy attack because it does not make any external calls during the execution of the withdraw() function. This means that the function cannot be interrupted by another contract, which is what allows reentrancy attacks to occur. As the Balance is updated before interacting with the other contract, it is not prone to a Reentrancy attack”[11].

Conclusion:

Reentrancy attacks are a serious threat to smart contracts. However, there are a number of ways to prevent them. As we talked about two important features “Reentrancy Guard” and “State Update”, it is very essential to take ongoing preventive measures to control Reentrancy Attacks. Developers must be aware of the possibility of reentrancy attacks and take precautions to avoid them. Developers may help safeguard their smart contracts from these threats by following the recommended practices. The future of reentrancy attacks is unknown. However, attackers are likely to continue to develop numerous ways to exploit this vulnerability. Developers must be very active and follow the newest security risks and take precautions to secure their smart contracts.

References:

[1] Darya Yatchenko - **“Reentrancy Attack”**
https://pixelplex.io/blog/smart-contract-vulnerabilities, Dec 15, 2022 .

[2] Neptune Mutual **- ”Block Timestamp Manipulation”**
https://neptunemutual.com/blog/understanding-block-timestamp-manipulation/,
Mar 16, 2023

[3] Immune Bytes **- “Front running attack”**
https://www.immunebytes.com/blog/front-running-attack/, Aug 24, 2022 .

[4] MacBobby Chibuzor **- ”TimeStamp Dependency attack”**
https://blog.logrocket.com/smart-contract-development-common-mistakes-avoid/,
Jun 17, 2022

[5] CoinDesk**-”DAO Attack”**
https://www.coindesk.com/learn/understanding-the-dao-attack/, Aug 11, 2022

[6] Rob Behnke **- ”Burger Swap Hack”**
https://www.halborn.com/blog/post/explained-the-burgerswap-hack-may-2021,
Jun 2, 2021

[7] Parity Technologies **- ” Parity Wallet Hack”**

https://www.parity.io/blog/a-postmortem-on-the-parity-multi-sig-library-self-destruct/, Nov 5, 2017

[8] MacBobby Chibuzor **- "Bank Smart contract"**
https://blog.logrocket.com/smart-contract-development-common-mistakes-avoid/, Jun 17, 2022

[9] Marish **- "Attack a Smart contract"**
https://coinsbench.com/understand-reentrancy-attack-by-building-a-bank-smart-contract-53ec44832402, Jan 27, 2022

[10] Doug Shipp **- "Secure a Smart contract"**
https://spin.atomicobject.com/2021/08/16/reentrancy-guard-smart-contracts/, Aug 16, 2021

[11] Nikhil Memane **- "Preventing Reentrancy with State Update"**
https://payatu.com/blog/reentrancy-attack-in-smart-contracts/ , Feb 27, 2023

[12]Ayman Alkhalifah **- "Reentrancy attack"**
https://www.frontiersin.org/articles/10.3389/fcomp.2021.598780/full **Dec 11, 2019**