Journal of Discrete Mathematical Sciences & Cryptography
ISSN 0972-0529 (Print), ISSN 2169-0065 (Online)
Vol. 26 (2023), No. 3, pp. 927–937
DOI : 10.47974/JDMSC-1779

Ensuring wallet application security by resolving reentrancy attacks in blockchain smart contracts

R. Manoj *
Department of Computer Science and Engineering
Manipal Institute of Technology, Manipal
Manipal
India

and

Manipal Academy of Higher Education, Manipal
Manipal University Jaipur
Jaipur
India

Sandeep Joshi †
Department of Computer Science and Engineering
Manipal University Jaipur
Jaipur
Rajasthan
India

Abstract

With the advancements in technology, blockchain systems have seen widespread use and rapid growth in the field of data security and verification. Blockchain is used in a variety of applications, including financial transactions, healthcare, insurance, Internet of Things, education, and many more, with the promise of increased skills and resilience. This smart distributed peer-to-peer design drew interest from a variety of businesses and communities outside the financial sphere. The major focus of the proposed work is on security challenges and limitations of Ethereum-based smart contracts. Ethereum smart contract is vulnerable to reentrancy security attack. The proposed work analyze reentrancy attacks and assess countermeasures to dissuade vulnerabilities on the network.

Subject Classification: *68M10.*

Keywords: *Security, Blockchain security, Reentrancy attack, Ethereum, Wallet application.*

* *E-mail:* **manoj.r@manipal.edu (Corresponding Author)**
† *E-mail:* **sandeep.joshi@jaipur.edu**

1. Introduction

Blockchain is a distributed ledger that uses cryptographic techniques for encryption of information and creates and validates data structures with algorithms. Data is stored in blocks and are connected in such a way that new blocks can be added and cannot be deleted. This new technology is started by Satoshi Nakamoto in 2009 when he created the first cryptocurrency bitcoin which is peer to peer open ledger [1]. Bitcoin successor Ethereum is a platform that supports external accounts and smart contracts also.

Cryptocurrencies are vulnerable to security attacks and these technologies are prone to security challenges [9][10][11]. One such attack is Reentrancy attack. Reentrancy attacks are a result of a contract performing an action before updating the corresponding state value; an example of this would be withdrawing ether from a bank before updating the user's account value. If the user has enough ether to withdraw the ether from the users account and then update the user's balance.

The purpose of this research was to identify effective strategies for dealing with the exploit identified in Smart Contracts built with Solidity. Based on the proposed system, it can be concluded that an effective way to prevent reentrancy attacks with the fix is demonstrated.

The paper is organized as follows. In section 2, we construct background of the work. Section 3 discusses a motivation that study different types of reentrancy attacks and existing defense mechanisms. Section 4 provides the system architecture of the proposed work and section 5 demonstrate the implementation details about fixing reentrancy attack. Finally, in section 5 we draw conclusions.

2. Background

A. *Ethereum*

Ethereum [2] is one of the popular blockchain platforms which can create algorithms written in general purpose programming languages allowing to develop simple applications ranging from wallet applications to complex financial applications. The programs are called smart contracts and transactions to Ethereum by the users can design smart contracts, call functions of contracts and/or transfer ethers.

B. *Solidity*

Smart contracts are coded in high level programming language solidity [3] and compiled to Ethereum Virtual Machine (EVM) byte code. First line of solidity code defines solidity version.

C. *Reentrancy attack*

Reentrancy attack in solidity [4] occurs when funds are withdrawn repeatedly from a smart contract and transferred to unauthorized smart contract till the funds get exhausted.

A reentrancy attack creates a recursive process that transfers funds between two smart contracts, the vulnerable contract and the malicious contract [5]. Figure 1 shows the attack scenario. Here are the steps of a re-entrancy attack:

1. The bad actor makes a call on the vulnerable contract, "X," to transfer funds to the malicious contract, "Y."
2. Contract X determines whether the attacker has the necessary funds, then proceeds to transfer the funds to contract Y.
3. Once contract Y receives the funds, it executes a callback function which calls back into contract X before the balance is updated.
4. This recursive process continues until all funds have been exhausted and transferred.

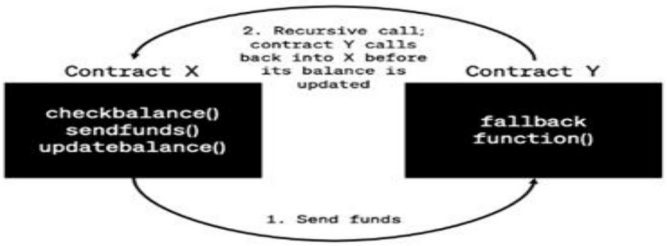

Figure 1

Reentrancy attack scenario

D. *Main Wallet Smart Contract*

```
pragma solidity 0.4.25; contract CashStore
{
int bal;
```

```
address public admin; constructor() public
{
bal = 100;
admin = msg.sender;
}
function getAdmin() public view returns (address) { return admin;
}
modifier onlyAdmin() { require(msg.sender == admin);_;
}
function getBalance() view public returns(int)
{
return bal;
}
function withdraw(int amt) public onlyAdmin
{
bal = bal - amt;
}
function deposit(int amt) public onlyAdmin
{
bal = bal + amt;
} }
```

3. Motivation

Reentrancy attacks are one of the common threats in Ethereum blockchain, which are associated with the Solidity programming language. The attacks occur when an adversary leverages an external call of a smart contract by forcing the contract to execute additional code by utilizing a fallback function to call back to itself.

There are two types of reentrancy attacks [5] single-function and cross-function attack. A single-function attack occurs when the adversary attempts to recursively call the vulnerable function. A cross-function attack occurs when the target function state is shared with another function that the adversary desires to exploit.

Current Defensive Methods

There are different methods used to protect smart contracts from reentrancy vulnerability. These proactive methods, which are utilized before the deployment of the smart contracts [7][12], are vulnerability-detection tools for Ethereum smart contracts [8], security based on

programming languages, and security based on the development of smart contracts [6].

4. System Architecture

The wallet application as shown in Figure 2 maintains information about your crypto balance while also allowing to conduct transactions via the Smart Contract and its security implementations from Attacks that have been mentioned.

The language which we are using is solidity and we have to mention the version number before we start writing the smart contract. We have to mention the name of the contract so in this case, we will be naming our contract "bank ". In the contract bank class, we make our functions in this case being deposit, withdraw and balance of type uint. Every function has an access specifier of public. We will also be using the view modifier for the balance function as it is just fetching a value. After completing the contract, we compiled it and is followed by its deployment. The smart contract is then connected to a test network by selecting the Injected web3 option for the environment. And now the contract can be deployed on the test network (ganache or ropsten). A suitable front-end UI is made and then connected to contract so that it becomes easier for the end user to conduct their desired tasks.

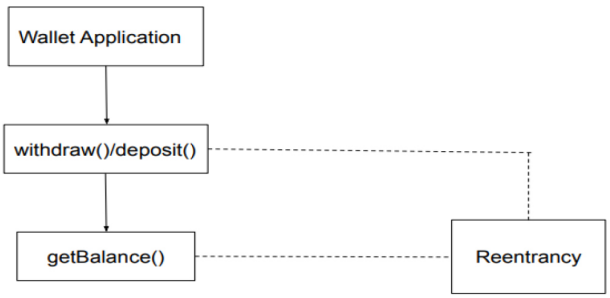

Figure 2

Architecture Diagram

5. Implementation

A. *Reentrancy Target Smart Contract*

While running the code will have a recursive loop that keeps going through and it never actually takes ether from the sender's balance because

this bank address stop balance is not the sender's address balance it is the balance of the smart contract. It is like a bank has a whole bunch of users there's a large amount that is the bank address balance however the users balance is only a small portion of that so what will happen is we go in and we keep recursively taking out ether until the bank is completely empty and now calling user has all the ether in the bank not just the ether they originally contributed. The reason that this works in this recursive loop is that because never actually get to the statement where we withdraw our amount from our balance every single time, we check that our withdraw amount is less than our balance it's always true so we're actually just taking ether from the hole and not ether from yourself until we finally run out of ether.

```
pragma solidity ^0.4.2; contract Target {
mapping (address => uint) private balances; address public owner;
constructor() public { owner = msg.sender;
}
function deposit() public payable {
require((balances[msg.sender] + msg.value) >= balances[msg.sender]);
balances[msg.sender] += msg.value;
}
function withdraw(uint withdrawAmount) public {
require(withdrawAmount <= balances[msg.sender]); require(msg.
sender.call.value(withdrawAmount)()); balances[msg.sender] -=
withdrawAmount;
} }
```

B. *Reentrancy Attack Smart Contract*

One is a vulnerable smart contract and the second one interacts with that smart contract so that we can exploit it and take all the money from the contract. In the smart contract that we're going to exploit, we have a deposit and a withdraw function. The deposit function allows us to get money into the smart contract in the form of ether. We have also contract naming which in other programming languages are like classes and also has a couple of variables. In order to interact with this contract, we have another smart contract.

In this smart contract we have created an interface to the target. We have done that by copying the function definitions. First thing is to put the pragma on top, this is something that needs to be at the top of every solidity contract, and it says what version we're using. We have made the target interface since we're interfacing with the target and then we have pasted the function definitions from the target which includes the deposit function which allows us to interact with the deposit function and the withdraw function too.

We deploy the contracts to a local JavaScript VM. Once we deploy it we will get a target address we should copy into the target address placeholder since that will be the address we need to target the smart contract. We will set up the bank address and then we'll use our target interface to call that bank address and we will set it to the address of the contract that we just deployed which would usually be an address on the blockchain. We set up the deposit function so that we can deposit ether inside of our deposit function. We're going to use our bank address interface we created and we're going to use the deposit function from the interface we're going to use the value of the amount that we have mentioned in the code which is the one ether. We then created a function that will get the balance that we just deposited to the account named getTargetbalance and we're going to make sure that that returns our value and that it's a public function. Then we return a unit value of our ether so we'll say return and we'll say our address, our address is going to be the bank address and then we use balance to get the balance of the contract and then from there we can deploy.

Then we are adding some extra functions, first function is the attack function, all this does is use the bank interface and withdraw the amount the amount is the one ether, the next function will automatically be called when somebody's sending money to us so we don't even need a name for this function we're going to mark it payable so we can accept money and then we're going to say if the bank addresses balance is less than the amount where that we're asking for which is 1 ether then we're going to call the withdraw function again and this is going to create our recursive loop. First, we save it and then redeploy, and we'll see our new retrieve stolen funds so what we do is we're going to deposit our amount through its recursive loop ran this attack and while never being updated to our balance we're able to steal everybody else's money.

```
pragma solidity ^0.4.2; contract targetInterface{
function deposit() public payable;
```

```
function withdraw(uint withdrawAmount) public;
}
contract simpleReentrancyAttack{ targetInterface bankAddress =
targetInterface(0xd9145CCE52D386f254917e481eB44e9943F39138); uint
amount = 1 ether;
function deposit() public payable { bankAddress.deposit.value(amount)
();
}
function getTargetBalance() public view returns (uint){ return
address(bankAddress).balance;
}
function attack() public payable { bankAddress.withdraw(amount);
}
functionretrieveStolenFunds()public{msg.sender.transfer(address(this).
balance);
}
function () external payable {
if (address(bankAddress).balance >= amount){ bankAddress.
withdraw(amount);
}
}
}
```

C. *Reentrancy Fix Smart Contract*

The first thing we will do is create a new file. In this smart contract file, we're going to create two functions, the first function is going to return the tx.origin address and the second function is going to return the msg.sender address of whoever calls this contract that way we can illustrate the differences and we'll understand why we can create a man in the middle attack on the blockchain using a malicious smart contract rather than confuse the matter with a more complicated example.

We'll create a contract and in here we're going to create two functions so the first function is going to be return tx address it's going to be public and it will return an address then we'll create an address variable and it will return the tx.origin so now what we're going to do is just copy paste this because the other functions can be basically the same thing except that it's going to return a msg.sender and since we're already familiar with msg.sender that the message.sender is a person who actually called this contract now what we need to do is create a smart contract that calls these functions so we can see the difference between msg.sender and tx.origin in reality this would be our malicious contract that we're fishing a user

with but in our case it's just going to be two simple functions that call these functions to show the difference so we create a new file and we'll go through the code anyway so we have an interface that we set up kind of like with our reentrancy attack where all we do is we copy paste the function definitions from hello and we'll replace public with external so now we have our target interface and we can use that target interface to link it up with the address in the blockchain then we add the address and then we will call it hellointerface = target interface(of the address).

You could create a malicious smart contract that convinces a user to take an action and when they actually take that action it passes through to the contract that maybe they only have authorization on, for example, with the authorization you had to kill function like maybe we wanted to kill that contract but we didn't have authorization to do it so we could create a contract to convince a user to make a transaction through our smart contract that then passes on their credentials into the other contract to kill it if the check was using tx.origin instead of msg.sender so that's where the vulnerability lies. The vulnerability lies right on this line, the line where we're actually using msg.sender or tx.origin. So if we do something like owner equals tx.origin we now have a vulnerability because it's actually checking the wrong thing it's not checking who called it checking the original account regardless if it was passed through.

```
pragma solidity ^0.4.2; contract Target {
mapping (address => uint) private balances; address public owner;
bool internal locked; modifier noReentrant() {
require(!locked, "No re-entrancy"); locked = true;
_;
locked = false;
}
constructor() public {
owner = msg.sender;
}
function deposit() public payable {
require((balances[msg.sender] + msg.value) >= balances[msg.sender]);
balances[msg.sender] += msg.value;
}
function withdraw(uint withdrawAmount) public noReentrant {
require(withdrawAmount <= balances[msg.sender]); require(msg.
sender.call.value(withdrawAmount)()); balances[msg.sender] -=
withdrawAmount;
}
}
```

6. Performance Analysis

To measure efficiency of security and performance in fixing the reentrancy attack, we performed test case execution in Hyperledger caliper, which is used for blockchain performance marking. The performance metrics chosen are throughput and latency. The test is conducted by varying the request rate. The throughput graph and latency graph as shown in Figure 3 and 4, the transaction number is plotted in X axis and transaction per second in y axis.

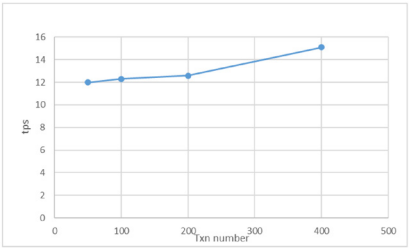

Figure 3
Throughput

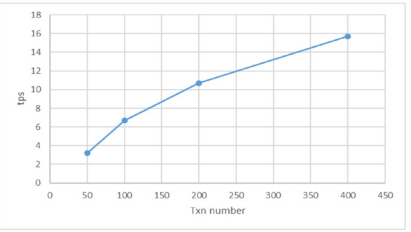

Figure 4
Latency

7. Conclusion

Proposed system identifies effective strategies for dealing with the reentrancy identified in Smart Contracts built with Solidity. Despite the negative connotations, exploits like these are important for the improvement for Smart Contract and Solidity development in general. Performance analysis is done for the fix of reentrancy attack by the metrics throughput and latency. Future exploration into contract modification techniques could be useful to finding further susceptibilities like these.

References

[1] S. Nakamoto, “Bitcoin: A Peer-to-Peer Electronic Cash System”.

[2] “Ethereum home page,” last accessed on 09-12-2022. [Online]. Available: https://www.ethereum.org.

[3] “Solidity home page,” last accessed on 12-12-2022. [Online]. Available: https://solidity.readthedocs.io/en/v0.5.1/.

[4] A. Alkhalifah, A. Ng, P. A. Watters, and A. S. M. Kayes, "A mechanism to detect and prevent ethereum blockchain smart contract reentrancy attacks," *Frontiers in Computer Science*, vol. 3, p. 1 (2021).

[5] Samreen, N. F., and Alalfi, M. H. "Reentrancy vulnerability identification in Ethereum smart contracts," The institute of electrical and electronics engineers, Inc.(IEEE) conference proceedings, London, ON, February 18, 2020 (IEEE), 22–29 (2020).

[6] Alexander, M., Markus, F.: Security vulnerabilities in ethereum smart contracts. In: Proceedings of the 20th International Conference on Information Integration and Web-based Applications & Services, pp. 375–380. ACM, New York (2018).

[7] Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., and Alexandrov, Y. "SmartCheck: static analysis of Ethereum smart contracts," The institute of electrical and electronics engineers, Inc.(IEEE) conference proceedings, Gothenburg, Sweden, May 27–June 3, 2018 (IEEE), 9–16 (2018).

[8] Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Buenzli, F., and Vechev, M. "Securify: practical security analysis of smart contracts," in 2018 ACM SIGSAC conference on computer and communications security, Toronto, ON, October 15–19, (ACM), 67–82 (2018).

[9] Alkhalifah, A., Ng, A., Chowdhury, M. J. M., Kayes, A. S. M., and Watters, P. A. (2019). "An empirical analysis of blockchain cybersecurity incidents," in 2019 IEEE Asia-Pacific conference on computer science and data engineering (CSDE), Melbourne, Australia, December 9–11, (IEEE), 1–8 (2019). doi:10.1109/CSDE48274.2019.9162381.

[10] Rita Roy, Kavitha Chekuri, G. Sandhya, Surya Kant Pal, Subhodeep Mukherjee & Nitish Marada. Exploring the blockchain for sustainable food supply chain, *Journal of Information and Optimization Sciences*, 43:7, 1835-1847 (2022), DOI: 10.1080/02522667.2022.2128535.

[11] Pawan Kumar, Reshma, Kamal Kant Sharma & Abhishek Gandhar. A block-chain excellency in the mining of crypto currency, *Journal of Information and Optimization Sciences*, 43:1, 123-130 (2022), DOI: 10.1080/02522667.2022.2037281.

[12] Anil Kumar, Ravinder Kumar & Sartaj Singh Sodhi. A novel privacy preserving blockchain based secure storage framework for electronic health records, *Journal of Information and Optimization Sciences*, 43:3, 549-570 (2022), DOI: 10.1080/02522667.2022.2042092.