Systematic Approach to Analyze Attacks on SCM: using Blockchain

B.Ratnakanth, K.VenkataRamana

Abstract: Smart contracts are programs, which are stored in a decentralized network i.e. Block chain. These are written by users to develop decentralized applications using different platform like Ethereum and bitcoin. In current scenario, even though blockchain support features like security and transparency. Because of solidity language vulnerability, there is a possibility of attacks on smart contracts in blockchain. So, to avoid those attacks like i.e. Locked Ether, Transaction order dependency and Time stamp dependency. We discussed, analyzed and tested these attacks in this paper. Further, in our project supply chain management for textile industry using block chain technology, we have developed smart contract using solidity language on Ethereum Platform. With the aim of protecting our project from these attacks, we are thoroughly and experimentally analyzed. And based on the experimental observations, we are going to protect our project from these attacks. All the above mentioned attacks are thoroughly studied and experimentally tested on Ganache, Ropston test network, Rinkeby test network using Remix IDE in JVM, injectedweb3and web3 provider environments. Finally, we have suggested security measures to protect from these attacks

Key words: blockchain, scm, attacks, smart ontracts, vulnerabilities

I. INTRODUCTION

A. Supply chain management (SCM)

The supply chain management (scm) is an innovative and very transparent, essential business process in every established organization now a day's [3]. The scm uses specific method to interact with customer's requirement with a supplier using a proper channel [4]. Supply chain process involves there are many other sub process. From the many years the technology developments have affected, particularly using software has impacted significantly on scm. These changes are occurred in more importantly in demand, operations, supply, warehousing and distribution [3]. For example, scm method has improved from material requirement planning (MRP) to manufacturing resource planning. And enterprise resource planning (ERP) to advanced supply chain planning (APS/APO) [3].We have been observing, there is tremendous improvement in the software technology to rationalize and development in supply chain process [3].

B. Block chain technology

The block chain technology was first conceptualized by satoshi Nakamoto for solving the double spending problem, which is inherent in all electronic transactions (nakamoto, 2008) [8].

Revised Manuscript Received on June 22, 2020.

B.Ratnakanth , Assoc. Prof, Vindhya Institute Of Technology and Science, Hyderabad

Dr.K.VenkataRamana, Asstt.Prof. Andhra University, Visakhapatnam

The block chain technology comprises of the core system Bitcoin, a digital currency that runs on Peer to Peer network without interference of any trusted third parties. [8].The block chain is an open, shared and distributed ledger, which stores the transactions in block chain, as conventional bank transactions are stored the in centralized ledger in the bank [2]. As block chain technology has unique characteristics, those are transfer of proprietary property, activity logging and access control. With these properties block chain enables the tracking of products and service flow across enterprise and borders. [8].

The block chain technology can be defined as decentralized database that provides very important properties. Those include immutability of stored transactions in block chain and it creates the trust among participants without involving third party, in a verifiable and permanent form [7]. One of the popular applications is exchange of digital assets among parties, also called as digital currencies i.e. Bitcoin, Ethereum, Litecoin (LTC), Ripple (XRP) .etc. They can be used not only for the digital assets, but also used for the execution of smart contract [9].

C. Smart contracts

The smart contracts are programs, those are facilitating to verify, enforce the negotiation and execution of legal contracts [12]. These contracts are executed on block chain through transactions and also interact with crypto currencies. When these contracts are deployed and run on block chain, smart contracts become autonomous entity [14]. Which are automatically performing some specific actions, when certain conditions are met. As smart contracts are run on block chain, they will run as programmed by the user without any censorship, downtime and third party interference [9].

D. Ethereum platform

The Ethereum is a public block chain based distributed computing platform, which facilitate smart contracts can run. It also provides decentralized Ethereum Virtual Machine (EVM), which is a run time environment to execute smart contracts over it [9].

E. Ethereum virtual machine (EVM)

The EVM performs computations and the state of smart contract, which supports stack based language with opcodes and related arguments [9]. So, smart contracts are a series of opcode statements. And those are sequentially executed by EVM.

F. Ethereum programming languages

Generally, the smart contracts are written in high level language, which are compiled to

EVM byte code. The high level languages are LLL (Low-level Lisp-like Language) [9]. Serpent (a Python-like language) [9], Viper (a Python-like language) [9], and Solidity (a JavaScript-like language) [9]. Where, LLL and Serpent were developed in the early stages of the platform, while Viper is currently under development, and is intended to replace Serpent. The most popular and widely used language is Solidity.

II. RELATED WORK

A. Block chain attacks

The smart contracts one of the main features is to manipulate and hold digital assets like Ethers. This feature makes the attackers to attack on smart contracts, even though the block chain known for security, immutability [10]. Hence in order to make sure our developed project supply chain management for textile industry is protected from these attacks. In previous paper we discussed about vulnerabilities in smart contracts i.e. DAO attack, Reentrance attack and Underflow attack [11]. In this paper we discuss about Locked Ether attack, Transaction Order Dependency attack and Time Stamp Dependency attack.

B. Locked ether (le) attack

Generally the smart contracts can have a payable functions withdraw i.e. which send Ether and deposit function i.e. which receive Ether. However, there are some instances, where withdraw function unable to send ether. There may be several reasons the **called** contract may not send Ether. One of the reason might be **calling** contract may depends on another contract, where the withdraw function defined [10]. Due to some reasons **called** contract has destructed, using the SELFDESTRUCT instruction of EVM i.e. the code has been removed and funds have been transferred [10] . For example the withdraw function may require another contract to send Ether. However, if the contract that relies has been destructed, then withdraw function will be unable to transfer the ether. Hence, funds are locked at the contract address. This was what happened in the **Parity Wallet bug** in November2017, locking millions of USD worth of Ether [10].

```
function deposit () public payable {
  require (msg.value==1 ether, "you can only send 1 ether");
 uint balance =address (this).balance;
  require (balance<=targetAmount,"game is over");
  If (balance== targetAmount) {
  winner=msg.sender;
  }
```

Problem: Participant of Ether game can deposit 1 Ether and become winner, and winner can claim 7 Ether, which is stored in the contract address. Likewise each participant can deposit 1Ether and he can claim for 7 Ether. Mean while, the Attacker deposits 5 Ether into contract address. After balance becomes prize amount 7 Ether. Attacker destroys the game contract, and he will lock 7 Ether at contract address using SELFDESTRUCT. Hence, he will break the game. So, no participant will win the game.

Deposit

The participant will deposit 1Ether into contract address to become winner of game.

***Algorithm 1*:** Deposit 1 Ether by each participant to become winner of the game.

***Input*:** *Target Amount* ← 7 Ether
Participant address $U_{addr1;}$
Participant address, $U_{addr2;}$
***Output*:** *balance of the contract address will be updated*

1: ***begin***
2: U_{addr1} ← *Fetch Participant1 address;*
3: U_{addr2}, Fetch Participant2 address;
4. *Function* ***Deposit*** *()*;
5: $D_{val\ 1}$ ← deposited 1 Ether in to Ether Game Contract Address. participant1;
5. $D_{val\ 2}$ ← deposited 1 Ether in to Ether Game Contract Address i.e. participant2;
6: update the balance;
7. ***End***

Claim reward

```
function claimReward() public {
  require (msg.sender==winner,"not a winner");
  (bool sent, )=msg.sender.call.value ( address(this).balance)("");
  require (sent, "Failed to sent ether");
  }
```

***Algorithm 2*: Claim for Reward by winner of the game**

Input: *Target Amount*, 7 Ether;
Participant1 address, $U_{addr1;}$
Participant2 address, $U_{addr2;}$
Contract address balance ← C_{bal} ;
Output: *winner of the Game will Receive 7 Ether;*

1. ***Begin*:**
2. U_{addr1}, *Fetch Participant1 address;*
3. D_{val} ← Fetch the 1 Ether value;
4. Function deposit ()
5. Update C_{bal} ←C_{bal}+ $D_{val\ 1}$
6. If (Bal ≥ 7);
7. If condition is true, then winner = msg.sender;
8. function claimReward ();
9. Else
10. Repeat step 2 to 6 for participant 1 and 2, until balance=7.
12. And declare the winner.
13. ***End***

Attacker

After each participant deposited 1 ether in to contract address, the attacker will know that total balance is 2 Ether. Then he will deposit 5 ether into contract address and update the balance to 7 Ether, then he will call **attack** function and he will destruct the game contract and forcibly lock the 7 ether at contract address. Hence no participant can access balance as contract was destroyed. As a result no participant will become winner.

```
Contract Attack {
  function attack (address payable target) public payable {
    selfdestruct (target);
  }
```

***Algorithm 3*:** Attacker to Lock 7 Ethers at contract address.

$C_{val} \leftarrow C_{val}$+ 5 Ether;

Input: *Attacker deposit 5 ether to contract address* , after Each participant deposit one Ether in to contract address.

Output: *target balance =7 ether and game is over. No winner and 7 ether locked at contract address.*

1. ***Begin*:**
2 call deposit ();
3. $C_{val} \leftarrow C_{val}$ + 5 Ether;
4. Update C_{val} = 7**;**
5. Game is over; no one is winner
5. Function attack ()
6. Function self-destruct ();
7. Ethers are locked at contract address. No participant can access the Ethers;
8. ***End***

Process flow

1. Each participant1 can call the **deposit** function and they will deposit 1 Ether in to contract address. And they will try to win the game.
2. If the current balance stored in the contract address is greater than or equal to 7 Ether, then the game is over.
3. After balance becomes 7 Ether, no one would send more ether.
4. If the current balance is 7 Ether, we set the winner is message sender.
5. The winner of the game can claim the reward by calling the function **claim reward**. This function will send all the ether to the winner. This is how the Ether game contract will works.
6. Suppose, attacker call **deposit** function and he will deposit 5 Ether in to contract address. And he makes the balance is 7 Ether.
7. As the sender is not participant. No one is winner of the game.
8. The **Attack** contract will call the **attack** function to break this game by using SELFDESTRUCT function. And he will forcibly send 7Eether into another address. .
9. If participant trying to deposit one more Ether, as balance becomes greater than seven .Game is over and winner is never set.

C. Transaction oreder dependancy attack

The basic functionality of Ethereum network is bundle the transactions to create a new block for every 17 seconds and getting confirmed. The role of miners is to receive the transactions and they will select which transactions to be included in a block. They will select generally based on who have consumed more gas. Further, whenever transactions are sent to Ethereum network, they are also forwarded to every node for further processing. Generally, the persons who is running Ethereum node will be able to know which transactions are being added in a new block. Hence there is possibility of race condition vulnerability in smart contracts, smart contract depends on the states of some of the following or previous smart contracts transactions. And it becomes attack, if there is change in the order of transaction included in the block.

***Problem*:** The Race condition which occurs on the Ethereum network is the race condition of ERC20 token standard. It contains a function called **approve,** which allows an address to spend token on their behalf.

The race condition happens in the network today is the race condition in the ERC20 token standard. The ERC20 token standard includes a function called **approve,** which allows an address to approve another address to spend tokens on their behalf. Assume that Alice has approved Eve to spend 'n' of her tokens, and then Alice decides to change Eve's approval to 'm' tokens. Alice submits a function call to approve with the value 'n' for Eve. Eve runs a Ethereum node, So Eve knows that Alice is going to change her approval to 'm'. Eve then submits a **transferFrom** request for sending 'n' of Alice's tokens to herself, but gives it a much higher gas price than Alice's transaction. The transfer from executes first so gives Eve ' n' tokens and sets Eve's approval to zero. Then Alice's transaction executes and sets Eve's approval to 'm'. Eve then sends those 'm' tokens to herself as well. Thus Eve gets n + m tokens even thought she should have gotten at most max (n, m).

I. Safe math

Safe Math is a library, which can be called by ERC20 contract to calculate mathematical operations.

```
library Safe Math {
function mul(uint256 a, uint256 b) internal pure returns (uint256) {
 if (a == 0) {
    return 0;
  }
  Uint256 c = a * b;
  require(c / a == b);
  return c;
 }
 function div (uint256 a, uint256 b) internal pure returns (uint256) {
  require (b > 0); // Solidity only automatically asserts when dividing by 0
  uint256 c = a / b;
   return c;
 }
  function sub (uint256 a, uint256 b) internal pure returns (uint256) {
  require(b <= a);
  uint256 c = a - b;
  return c;
 }
 function add (uint256 a, uint256 b) internal pure returns (uint256) {
  uint256 c = a + b;
  require(c >= a);
  return c;
 }
 function mod(uint256 a, uint256 b) internal pure returns (uint256) {
  require(b != 0);
  return a % b;
 }
}
```

Algorithm 4: **Safe Math** Library to perform mathematical operations

Input: A, B;
C←0;

Output: Update the value of C for math operations.

1. *Begin*:
2. A← Fetch the value;
3. B ← Fetch the value;
4. C←0,
5. Function mul ();
6. C=a*b;
7. Function div ();
8. C=a/b;
9. Function sub ();
10. C= a - b;
11. Function add ();
12. C = a + b;
13. Function mod ();
14. C = a % b;
15**. *End***

transfer

The **transfer** event called with parameters i.e. owner address, spender address and Ether value and it finds the hash value of these parameters .The hash value,spender address, ether values are stored in contract address..

```
function transfer (address to, uint256 value) public returns (bool) {
  require (value <= _balances [msg.sender]);
  require (to != address (0));

  _balances [msg.sender] = _balances [msg.sender].sub (value);
  _balances [to] = _balances [to].add (value);
  emit Transfer (msg.sender, to, value);
  return true;
 }
```

Algorithm 5: **Transfer of 5 ether from owner to spender**

Input: balance [spender] ← 0;
balance [owner] ← 10 ether;
allowed [owner] [spender]=0;

Output: Transfer the 5 ether to spender

1. Begin:
2. Owner ← 10 ether;
5. Function **transfer ();**
6. balance [Spender] ← 5 Ether
7. Balance [Owner] ← 5 Ether
8. Calculate←hash (owner, spender, value)
9 contract address ← hash
10. ***End***

Approve the transaction

The spender will submit the i.e. salt, value to the contract address. Then contract address calculates hash. And compare hash with stored hash. If both same are then spender can claim for amount. Otherwise contract address rejects amount claimed.

```
function approve (address spender, uint256 value) public returns (bool) {
  require (spender != address (0));

  _allowed [msg.sender][spender] = value;
  emit Approval (msg.sender, spender, value);
  return true;
 }
```

Algorithm 6: **Approve the transaction**

Input: spender address;
Owner address;
Salt;
Value ←5 Ether;
Output: approve the allowed [msg.sender, spender, value];

1. Begin:
2. function approve ();
3 ._allowed [msg.sender][spender] = 5 Ether;
5. emit approve ();
6 contract address ← hash [msg.sender,spender,value];
7. ***End***

Transferform.

The spender submits the **TransferForm** to node for authorization. The node will check the hash values and if satisfied, then spender can spend ether on behalf of owner.

```
function transferFrom (address from, address to, uint256 value) public returns (bool) {
  require (value <= _balances [from]);
  require (value <= _allowed [from][msg.sender]);
  require (to != address (0));

  _balances [from] = _balances [from].sub (value);
  _balances [to] = _balances [to].add (value);
  _allowed [from][msg.sender] = _allowed[from][msg.sender].sub(value);
  emit Transfer (from, to, value);
  return true;
 }
```

Algorithm 7: Submission of TransferForm by spender and get permission to spend the values on behalf of owner.

Input: spender address;
Salt;
Value;
Output: spender will be authorized to spend the ethers on behalf of owner;
1. ***Begin***:
2. Function TransferForm ();
3 spender ← authorized on behalf of owner;
4. emit transfer ();
5. ***End***

Process flow:
1. Assume that ALICE has approved EVE to spend 'n' number of tokens on behalf of her.
2. Later ALICE decides to change EVE's approval to 'm' tokens.
3. Initially ALICE submits a function call to approve value 'n' for EVE.
4. EVE runs an Ethereum node, so she knows that ALICE going to change 'm' tokens instead of 'n' tokens to spend on behalf of her.
5 Eve then submit **TransferForm** request for sending of 'n' tokens to herself. But gives much higher gas price than Alice.
6. The **TransferForm** request has been executed first, as gas price is very high. So' n' tokens transferred to EVE. And sets EVE's approval to zero.
7. After That Alice transaction executes and sets EVE approval to 'm'.
8. Eve then submit **TransferForm** request to receive 'm' tokens from Alice. EVE receives 'm' tokens as well from ALICE.
9. The EVE received total of m+n, even though she is only approved to get 'm' tokens

D. Timestamp dependence attack

The smart contracts are often access to time values to perform certain type of operations on Ethereum network. Those are **block.number** and **block.timestamp**. These parameters can give you a sense of current time or delta on network. However, they are not true values some times. The Time stamp dependency is vulnerability in block chain, which can be take advantage by the malicious miners.

The miners can be delayed or advance the time stamp by a few seconds. Generally, this value between 15 seconds after the current block time stamp. Because the new block have to be validated within the 15 second by miner. As miner will delayed the new block validation, it enables the Ethereum network will not be synchronized with global clock. For example, smart contract may utilize the current time stamp to generate random numbers for the purpose of determine the lottery result. Miners can take advantage of this, and they may change time stamp of the of new block**.** Hence the random generation can be altered [16].

Lock token for specific time period

***problem*:** The Token Will Be Locked for Specified Time Period.

This Smart contract is to lock the token for specific time period. So that token can not withraw until the locked period is completed. In this smart contract to compute the lock period, we have consider the block number as input.

```
// Tokens can be locked for exact time specified
  function locket (uint _time, uint _amount) public payable {
    require (msg.value == _amount, "must send exact amount");
    users [msg.sender].unlockBlock = block.number + ( _time / 14);
    users [msg.sender].amount = _amount;
  }
```

***Algorithm - 8*:** Tokens should be locked for specified time period.

***Input*:** _time;
_amount ← 5wei;
Locked period, Unblock =0;

Output: *token is locked*

1. Begin:
2. _time = 20sec;
3. Function lockEth ();
4. amount locked;
5. users [msg.sender].amount =_amount;
6. user [msg.sender]. unblock=_block.number + (_time /14);
7. ***End***

Withdraw token if lock period is over

The smart contract is developed to withdraw amount if the block period is completed.

```
// withdraw tokens if lock period is over
  function withdraw () public {
    require (users [msg.sender].amount > 0, 'no amount locked');
    require (block.number >= users [msg.sender].unlockBlock, 'lock period not over');
    //unlock condition
    uint amount = users [msg.sender].amount;
    users [msg.sender].amount = 0;
    (bool success, ) = msg.sender.call.value (amount) ("");
    require (success, 'transfer failed');
  }
```

***Algorithm - 9*:** Withdraw token if lock period is completed

***Input*:** wait until lock time period
Unblock=block.number + (_time/14);

Output: *withdraw amount*

2. If users [msg.sender].amount ≥ 0;
3. If (block. number ≥ users [msg.sender].unblock;
4 if true amount = users [msg.sender].amount;
5. users [msg.sender].amount=0;
6. ***End***

Process flow

1. The algorithm locks the 5 wei for Specific time period by specifying time period in seconds based on current block number.
2. Call the function **lockEth** ().
3. Initialize Time period based on current block number and amount in wei as input.
4. Check the conditions for amount in wei and Time period for lock ether=block.number+ (time/14).
5. Transfer the amount in to users account and locked.
6. Call the function **withdraw** ()
7. If lockperiodi over.
8. Check the condition if block.number is greater than previous block number
9 then withdraw amount.
10 else wait until lock period completed.

III. RESULTS AND ANALYSIS

We have experimentally tested all these attacks on solidity environment. We are practically examined and analyzed results on Ganache GUI, Ropston test network, Rinkeby test network in JVM, Injected web3 and web3 provider environment. We are tabulating the transactional information of each transaction and extracted the information like i.e. Gas, gas cost, blocks information and transaction time etc.

A.lock ether attack

Table - I: Experimental Results observed on Ganache Using web3 provider environment

LOCK ETHER ATTACK						
SNO	OPERATION PERFORMED	INPUT TO THE FUNCTION CALL	UPDATED BALANCE OF THE ETHER GAME CONTRACT (ETHER)	BLOCK NO	TRANSACTION TIME	TOTAL COST (EITHER)
1	ETHER GAME CONTRACT DEPLOYED	-	-	1	**2.55sec**	**0.00689544 ETH**
2	ATTACKER CONTRACTDEPLOYED	-	-	2	**2.54 sec**	**0.00182926 ETH**
3	DEPOSI 1 ETHERALICE	1 ETHER	1 ETHER	3	**2.24 sec**	**0.00047284 ETH**
4	DEPOSIT1 ETHER BY BOB	1 ETHER	2 ETHER	4	**1.83sec**	**0.00047284 ETH**
5	DEPOSIT 5 ETHER BY Attacker	5 ETHER	7 ETHER	5	**1.70 sec**	**0.00047284 ETH**
6	WINEER OF THE GAME	No one is winner as 7 ether locked at contact address	-	5	**1.37sec**	-
7	ADDRESS OF WINNER	0	**No one is winner of the game as 7 Ether locked at ether game contract address**			

Table - II: Experimental Results observed on Ganache Using web3 provider environment

SNO	OPERATION PERFORMED	INPUT TO THE FUNCTION CALL	UPDATED BALANCE OF THE ETHER GAME CONTRACT	TRANSACTION COST (GAS)	Gas price	TOTAL COST (EITHER)
1	ETHER GAME CONTRACT DEPLOYED	-	-	344772	20000000000	0.00689544 ETH
2	ATTACKER CONTRACT DEPLOYED	-	-	91463	20000000000	0.00182926 ETH
3	DEPOSIT1 ETHER BY ALICE	1 ETHER	1 ETHER	23642	20000000000	0.00047284 ETH
4	DEPOSIT1 ETHER BY BOB	1 ETHER	2 ETHER	23642	20000000000	0.00047284 ETH
5	DEPOSITED 5 ETHER BY ATTACKER	5 ETHER	7 ETHER And the etherLocked at contract address and ethergame contract destryoed	13309	20000000000	0.00026618 ETH
6	WINEER OF THE GAME	No one is winner as 7 ether locked at contact address	-	-	-	-
7	ADDRESS OF WINNER	No one win the game as 7 ether locked at ether game contract address 0				

Figure 1. Output observed for Lock ether attack on REMIX IDE using web3 provider.

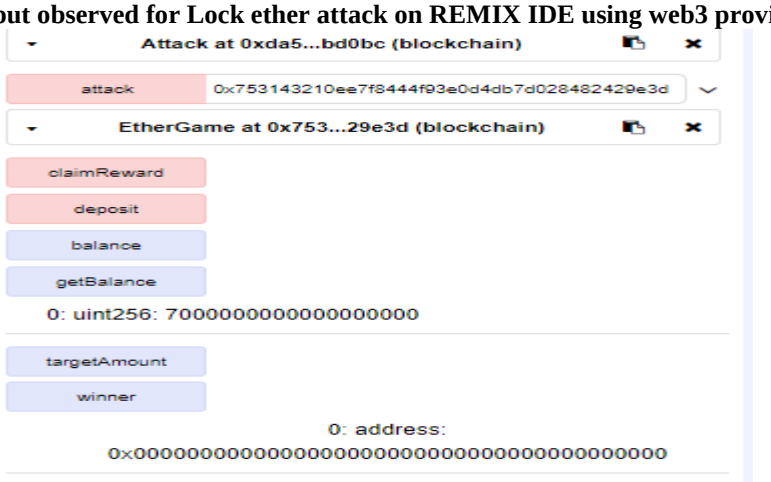

Figure: output observed for Lock ether attack on Remix IDE.

***Security measures*:** We can protect the smart contract against lockether attack by declaring balance state as public and update the balance for every time we call the deposit function. So that attacker cannot deposit amount in unauthorisely. The following conditions will protect the smart contract from attacker.

Conditions-1: Declare uint balance public;
Condition: 2 balance [address]+= msg.value**;**

B. Transaction order dependency attack

Table - III: Results observed on Rinkeby Test network using injected web3 environment

SNO	OPERATION PERFORMED	INPUT TO THE FUNCTION CALL	UPDATED BALANCE	TRANSACTION COST (GAS)	TOTAL COST ETHER	TRANSACTION TIME	BLOCK NO
1	CONTRACT DEPLOYED	20 wei	-	673492	0.000673492 Ether	28.97 sec	6820078
2	DEPLOY SELFMATH LIBRARY	-	-	69864	0.000069864 Ether	22.79 sec	6820116
3	BALANCE OF OWNER	20 wei	20 wei	-	-	-	-
4	APPROVE	SPENDER,10 wei	10 wei	43947	0.000043947 Ether	37.99 sec	6820209
5	SPENDER	-	10 wei	-	-	-	-
6	TRANSFERFROM	OWNER,THIRDPERSON, 5 wei	OWNER=15 wei SPENDER=5 wei THIRD PERSON=5 wei	59615	0.000059615 Ether	41.23 sec	6820321

Table - IV: Results observed on Rinkeby Test network using injected web3 environment

SNO	OPERATION PERFORMED	INPUT TO THE FUNCTION CALL	UPDATED BALANCE	Block mined	BLOCK NO	TRANSACTION TIME	NO OF TRANSACTIONS PER BLOCK	BLOCK SIZE
1	CONTRACT DEPLOYED	20 wei	-	15 sec	6820078	28.97sec	12	8,837 bytes
2	DEPLOY SELFMATH LIBRARY	-	-	15 sec	6820116	22.79sec	12	2,652 bytes
3	BALANCE OF OWNER	20 wei	20 wei	-	-	-	-	-
5	APPROVE	SPENDER,10 wei	10 wei	15 sec	6820209	37.99 sec	6	5,266 bytes
4	SPENDER	-	10 wei	-	-	-	-	-
6	TRANSFERFROM	Owner,Third person,5wei	Owner=15 wei Spender=5 wei Third person =5 wei	15 sec	6820321	41.23sec	9	2,554 bytes

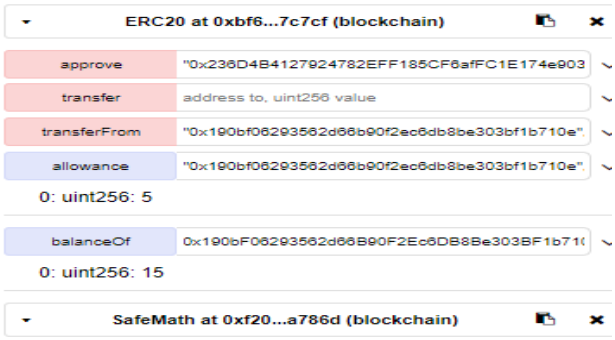

Figure 2. Output after execution of smart contract on RINKEBY TEST Network

***Security Measures*:**

The security measure for race condition is submission of information in exchange of reward is commit reveal hash scheme. Instead of submitting value to the party, submit hash of (salt, spender, value). The contract stores this hash as well as sender address. To claim reward the client has to submit salt and value. Then contract address generates hash from (salt, value, msg.sender) and compare the hash produced with the stored hash. If the hash matches contract release the reward. So, instead of submitting clear text information, we have to submit hash values to the blockchain. So, miner's i.e attacker can not understand the hash values. Hence we can prevent transaction order dependency attacks.Whenever transactions involved amount or secret information; we can implement this technique to protect from attacker .i.e malicious miner.

The best fix for the ERC20 race condition is to add a field in approve function i.e. expected current value. If Eves current expecting value is not Alice indicate, then revert approval.

C. Time stamp dependency attack

Table - V: Results observed on ROPSTON TEST network using injected web3 environment

TIMESTAMP DEPENDENCE ATTACK								
SNO	**OPERATION PERFORMED**	**INPUT TO THE FUNCTION CALL**	**Block mined in**	**BLOCK NO**	**TRANSACTION TIME**	**NO OF TRANSACTIONS PER BLOCK**	**BLOCK REWARD ETHER**	**BLOCK SIZE**
1	**CONTRACT DEPLOYED**	-	7 secs	8258753	20.94SEC	**27**	2.044367274499 981 Ether	19,904 bytes
2	**TOKEN LOCKED PERIOD**	Time=20 sec amount=5wei	17sec	8258867	29.25sec	**16**	2.010942032599 981 Ether	3,355 bytes
3	**WITHDRAW TOKEN IF LOCKED PERIOD IS OVER**	-	10 secs	8258871	20.43 sec	10	2.0103968333 Ether	24,705 bytes

Table - VI: Results observed on ROPSTON TEST network using injected web3 environment

SNO	OPERATION PERFORMED	INPUT TO THE FUNCTION CALL	UPDATED BALANCE OF THE ETHER GAME CONTRACT	TRANSACTION COST (GAS)	Gas	TOTAL COST (EITHER)
1	**CONTRACT DEPLOYED**	-	-	312717	312717	0.014384982 Ether
2	**TOKEN LOCKED PERIOD**	Time=20 sec amount=5wei	-	61842	61842	0.002844732 Ether
3	**WITHDRAW TOKEN IF LOCKED PERIOD IS OVER**	-	5 wei	21623	38832	0.000994658 Ether

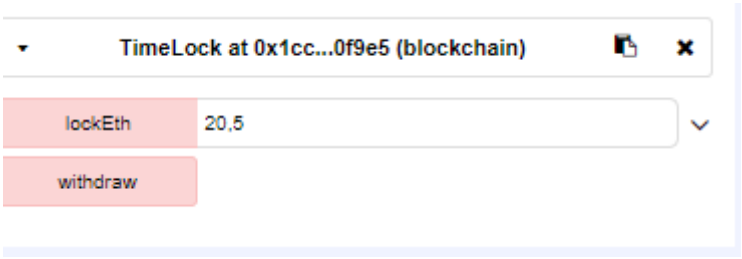

Figure 3: Output after execution of smart contract on ROPSTEN TEST Network

Security Measures:

However, miner cannot set a time stamp of new block, not less than current block time stamp and not too far above the current block time stamp i.e maximum difference between successive blocks are 17 sec. By considering all the above information, the developers can not rely on the block time stamp.

IV. CONCLUSION AND FUTURE SCOPE

From the above discussion and experimental observations, we found that lock ether attack is occurred due to the balance related states are not properly declared as public and not updated immediately after changes occurs in the function definition. Similarly, in transaction order dependency attack, we should provide important information to Ethereum node in the form of hash code, rather than in the form of conventional format. Finally, time stamp dependency attack can be avoided by not using time stamp of the block in the block chain. Because the miner can change the time stamp of particular block in the block chain. After examine these attacks, all these attacks will seriously damage the smart contracts, If the smart contract contains the Ether related operations, We must protect our smart contract against these attacks. Otherwise, malicious user or miner will make use of these vulnerabilities and he will be benefited. As a future work, we are going to proectt our project supply chain management for textile industry by applying secure solutions of above attacks

REFERENCES

1. Satchain: Secured Autonomous Transactions in Supply Chain using Block Chain, International Journal of Innovative Technology and Exploring Engineering (IJITEE) ISSN: 2278-3075, Volume-9 Issue-6, April2020,B.Ratnakanth,Dr.K.venkataRamana .https://www.ijitee.org/download/volume-9-issue-6/ .
2. Secure payment system in supply chain management using blockchain technologies, by B.Ratnakanth, dr.k.venkataramana.www.jetir.org (issn-2349-5162),© 2019 jetir june 2019, volume 6, issue 6,
3. Block chain Technology: Supply Chain Insights from ERP, Arnab Banerjee, Advances in Computers # 2018 Elsevier Inc. ISSN 0065-2458 All rights reserved. https://doi.org/10.1016/bs.adcom.2018.03.007.
4. Supply chain and logistics controller – two promising professions for supporting transparency in supply chain management**,** ISSN: 1359-8546, Publication date: 13 April 2020.
5. The impact of the blockchain on the supply chain: a theory-based research framework and a call for action, Horst Treiblmaier ISSN: 1359-8546, Publication date: 10 September 2018.
6. Block chain in Logistics and Supply Chain: a Lean approach for designing real-world use cases, Guido Perboli1,2,3, Member, IEEE, Stefano Musso1,2, Mariangela Rosano1,2DOI 10.1109/ACCESS.2018.2875782, IEEE Access. 2169-3536 (c) 2018 IEEE
7. Land records on Blockchain for implementation of Land Titling in India, Vinay Thakura, M.N. Dojab, Yogesh K. Dwivedic, Tanvir Ahmadd, Ganesh Khadangae
8. Supply chain re-engineering using blockchain technology: A case of smart contract based tracking process, Received in revised form 21 March 2019; Accepted 26 March 2019 . 0040-1625/ © 2019 Elsevier Inc. All rights reserved.

9. Smart Contracts: Security Patterns in the Ethereum Ecosystem and Solidity, Maximilian Wöhrer and Uwe Zdun 978-1-5386-5986-1/18 c , 2018 IEEE
10. SmartContractVulnerabilities: DoesAnyone Care? DanielPerez, BenjaminLivshits ImperialCollegeLond, arXiv:1902.06710v3 [cs.CR] 17 May 2019
11. Security Analysis Methods on Ethereum Smart Contract Vulnerabilities — A Survey, Purathani Praitheeshan?, Lei Pan?, Jiangshan Yu†, Joseph Liu†, and Robin Doss?. arXiv:1908.08605v2 [cs.CR] 9 Jan 2020.
12. Smart Contract: Attacks and Protections, sarwar sayeed , hector marco-gisbert (senior member, ieee), and tom caira, Received December 6, 2019, accepted January 17, 2020, date of publication January 30, 2020, date of current version February 10, 2020.*Digital Object Identifier 10.1109/ACCESS.2020.2970495*
13. Security Vulnerabilities in Ethereum Smart Contracts, Alexander Mense†, Markus Flatscher, iiWAS '18, November 19–21, 2018, Yogyakarta, Indonesia © 2018 Copyright is held by the owner/author(s). Publication rights licensed to ACM.
14. .Finding the greedy,Prodigal,and suicidal contracts at scale, Ivica Nikolic ,Ashish kolluri, Ilya Surgey,Prateek suxena,Aquinas hobor, arXiv:1908.08605v2 [cs.CR] 9 Jan 2020.
15. On Blockchain Security and Relevant Attacks, Joanna Moubarak,Eric Filiol,Maroun Chamoun, 978-1-5386-1254-5/18/$31.00 ©2018 IEEE.
16. Defects and Vulnerabilities in Smart Contracts, a Classification using the NIST Bugs Framework

AUTHORS PROFILE

B.Ratnakanth: He is working as Assoc. Prof, in VITS, Hyderabad. He is graduated B.Tech from SRKREC, Bhimavaram,A.P And He has completed M.E from SRTMU, Nanded, M.S. He has 17 years teaching experience. His Interest areas are security,image processing , computer networking.

Dr.K.VenkataRamana: He is working as Asst.Prof in Andhra University, He is graduated B.Tech from Nagarjuna University. And, he is completed M.Tech from Andhra University. He is awrded Ph.D from Andhra University. He has 17 years teaching Experience. His Research interest is Security, Machine Learning, and Artificial Intelligence.