Efficient Methods to Avoid Smart Contract Vulnerabilities Using Block Chain

B. Ratnakanth[1], M. Sahiti[2], Dr. K. Venkata Ramana[3]

[1]Vignan Institute of Technology & Science, Hyderabad, A.P., India

[2]Dep: Computer Science and System engineering, Andhra University, Visakhapatnam, A.P., India

[3]Assistant Professor, Andhra University, Visakhapatnam, A.P., India

Abstract: ***Ethereum smart contracts are programs which will run inside public distributed network called block chain. These smart contracts are used to perform operation over ether i.e transfer, receiving across the blockchain, by public to manage their accounts. These smart contracts are immutable once deployed on blockchain. So, developers need to make sure that smart contracts are bug- free at the time of deployment. As we are developing supply chain management (SCM) for textile industry project, to protect the project from smart contract vulnerabilities. In this paper we have analyzed the Decentralized Autonomous organization i.e DAO attack, which takes the advantage of smart contract vulnerability. Some functions are exposed to access by external contracts. The attacker makes use of vulnerability in smart contract and he can implement code to recursively call the function to transfer the funds in to his own account. And also we analyzed Reentrancy attack, which also used by attacker to recursively call the contract to multiple transfers of funds to his own account. And finally we analyzed Underflow attack, which make use of vulnerability in smart contract while transferring ethers between the users without considering limitations of integers values i.e uint8,uint16 etc.***

Keywords: attacks, blockchain, smart contract, supply chain management

1. Introduction

The fast digitization of industry in supply chain management. Opportunities around digitization have made possible for supply chain to able to access, store and process huge amount of data from the firm and also externally. For instance, the manufacturing industries are now able to obtain customer data to personalize the sales process, product design and service. The amount of data stored and distributed also improved in both forecasting accuracy and development of predictive solution [G. Schniederjansa, Carla Curadob, Mehrnaz Khalajhedayatia in 2019].

Block chain technology: From the invention of Bitcoin, a crypto currency, in 2008, Blockchain technology has placed in the central point of interest among a diverse range of researchers and developers [6]. The Blockchain is a decentralized ledger, which stores all the transactions made on peer to peer network. Block chain technology is secure, open source and immutable.

The main advantage of block chain technology over a other technologies is that it enables the users can make transactions securely without interference of any intermediary [mohammad dabbagh, mehdi sookhak2, and nader sohrabi safa3]. The blockchain is become popular now a days in industries finance, IOT, health care, and supply chain management system. [6].

Smart contract concept was initially proposed by Nick Szabo in 1997. A smart contract is a program that runs on the block chain autonomously. Smart contract, which enforce the pre-defined rules of an agreement without the interference of trusted third party [7]. This feature support the smart contract with low transaction cost, but there is a security issue such as it has inherent immutability of blockchain i.e. not possible to change the contract, once it is deployed in the block chain

Smart contracts are referred as self- autonomous and self-verifying agent, consist of fields and functions. The deployed smart contract receives a contract account address, which is different from user accounts, who are interact with smart contract. The smart contracts are converted into low level byte code called as “Ethereum virtual machine code” or EVM code. As Ethereum is a public block chain, so byte code of every smart contract is publicly available and every node in the block chain can see the code. So, the behavior of smart contract is predictable. The smart contracts have a functionality to hold a state, exchange digital assets, store data, receive the information from external contracts. Smart contract function is triggered by either message call or transactions sent to the contract unique address [10].

Call to the Unknown

Some of the primitives used in solidity to invoke functions and to transfer ether may lead to the side effects of invoking the **fallback** function of the callee/recipient. The CALL invokes a function (of another contract or itself), and transfer ether to the recipient. **Ex:** One can invoke a function test of contract x as follows

x. call.value (amount)(bytes4(sha3(“test(uint256)”)),n);
Here the called function is identified by first 4 bytes of its hash signature, amount refer to the how many wei needs to be transferred to x, and n represent the actual parameter of test function. Suppose, if a function with this signature is not available in the recipient contract x, then the fallback function of x is executed [9].

Here is a method to forward gas to the receiving contract using addr.call.value () (“”). It is basically same as addr.transfer(x), only that it forwards all remaining gas and make possible the recipient to perform more exclusive actions. There might be calling back into the sending contract or other state variable changes you might not have thought of. Finally, it allows more flexibility for malicious users [10].

If a contract receives ether (without a function being called), either the receive Ether or the fallback function is executed. If it does not have a receive nor fallback function, the Ether will be rejected (by throwing an exception). During the execution of one of these functions, the contract can rely on the “Gas stipend” it is passed (2300 gas) being available to it at that time [10].

Fallback Function: A contract can have at most one fallback function, declared by fallback () external payable. This function cannot have arguments, cannot return anything and have external visibility. It is executed on a call to the contract, if none of the other functions match the given function signature, or if no data is supplied at all and there is no receive ether function

2. Methodology

2.1 DAO ATTACK

The DAO is abbreviated as Decentralized Autonomous Organization. The concept behind the DAO in block chain generally, is to codify the rules and regulations of organization in the form of smart contracts. Thus, eliminating the use of documents and administration by individuals and to create autonomous system, in the form of decentralized control. The group of people writes the smart contract to govern the organization. Then followed by initial funding period, where the participants purchase the tokens in exchange of ether, to get the voting rights. Followed by, the people can submit project proposals to DAO, and they can approved by the members, who has voting rights to get funds from DAO. [1]. The DAO came into operate in 30^{th} April 2016, with an amount of initial funding period for 28 days. By that period, DAO has collected \$150 million in ether from 11,000 participants. From that day onwards attacker trying to attack DAO to hack the funds. On 17th June, an attacker exploited the Re-entrancy vulnerability and he managed to drain 3.6 million ether from DAO [1].

Problem: DAO ATTACK
The main contract contains DAO funds deposited by users with their address. The users can deposit and withdraw their funds through their address. But attacker create a Malicious contract and deposits money in to DAO and attack the DAO to drain all the funds by using fallback () function and flaw in msg.sender.call.value (amount) (“”);
Sol:

```
function donates (address to) external payable {
    credit [to]+=msg.value;
  }
```

Fig: deposit function

Algorithm1: DEPOSIT IN DAO CONTRACT

Any user can deposit ether **into DAO contract through their address and update balance**
Input:
$Uaddr_1$ ← User Address1
U_{bal1} ← update balance Uaddr1
D_{usr1} ← deposit into $Uaddr_1$

Output: update the balance of the user
1. **Begin**: $Uaddr_1$ ← Fetch user address1
2. Call **deposit()** function
2. D_{usr1} ← deposit **20** ether into DAO
3. U_{bal1} ← update balance of user1
4. **End**

```
functions withdraw (uint amount) external {
    If (credit [msg.sender]>=amount){
      msg.sender.call.value (amount)();
      credit [msg.sender]-=amount;
    }
```

Fig: DAO contract withdraw function

```
function () payable external{
   dao.withdraw (dao.queryCredit (address (this)));
  }
  function getJackpot () external {
     owner.transfer (address (this).balance);
   }
```

Fig: fallback () function and jackpot () function

Algorith2: **ATTACKER CONTRACT**
Attacker can deposit ether into DAO. And attack the DAO by Recursive fallback () function.

Input:
$Uaddr_2$ ← Attacker Address2
$Uaddr_3$ ← Attacker contract Address3
U_{bal3} ← Attacker contract balance $Uaddr_3$
D_{usr32} ← deposit into Uaddr3

1 Begin: $Uaddr_2$ ← Fetch attacker address2
2. $Uaddr_3$ ← Fetch attacker contract address3
3. Call **deposit()** function
4. D_{usr3} ← Attacker deposited **10** Ether into DAO through Malicious contract address
5. U_{bal3} ← update balance of Attacker contract address
6..Attacker call **Fallback ()**
8. **Withdraw ()** function called in side fallback() function.
9. Withdraw () function called recursively until 30 ether transferred to Malicious contract address.
9. Attacker call **getJackpot ()**
10 Funds 30 Ether transferred to Attacker account From malicious contract address.

2.2 Under Flow Attack

In solidity language there is a value limitation exist for integers, lack of awareness of these limitations lead to some wrong results. In solidity language an integer data are represented with bit level specification, such as **uint8** used for 8-bit unsigned integer or **uint,** which is an alias for **buint256** used to represent 256 bit unsigned integer. The bit level specification of integer leads to value storage limitations. Like, when performing operations such as addition, subtraction there will be overflow / under flow can occurs [2].

Problem:
Bank transaction between sender and receiver

```
function contribute () public payable {
 balances [msg.sender] = msg.value;
 }
```

Fig: **contribute ()** function

```
function getbalance () view public returns (uint){
 return balances[msg.sender];
 }
```

Fig: getbalance () function

```
function transfer ( address _receiver, uint _value) public payable{
//require (balances [msg.sender]- _value>=5);
//require conditionde to avoid any underflow in the program
 Balances [msg.sender] = balances [msg.sender]- _value;
 Balances [_receiver] = balances [_receiver]+ _value;
}
```

Fig: Funds **transfer ()** function

Algorithm1: contribution to the sender account

Input:
$Uaddr_1$ ← sender account
D_{addr1} ← sender account
V_{al} ← value to be transfer
D_{addr2} ← receiver account
B_{bal1} ← Balance of the sender account
B_{bal2} ← receiver account

Output:
1. Begin:
2. Fetch sender account ← $Uaddr_1$
3. Contribute to the sender account **contribute() function**
4. V_{al} ← value to be transfer
5. Fetch Daddr2 ← receiver account
6. Call **transfer ()** function
7. Update the balance of sender and receiver account
8.Chek the **underflow/over flow attack**
9. Repeat step 1 to 8 to observe different cases

2.3 Reentrancy Attack

The main danger of calling an external contract is that they can take over the control flow and make changes your data that the called was not expecting. In Reentrancy attack (i.e recursive call attack), a malicious contract can calls back to the calling contract before the first invocation of the function is finished. This may lead to the different invocation of the function to perform in a undesirable manner. The function could be called repeatedly, before the first invocation of the function was finished [4].

External calls:
When call made to untrusted contracts can always introduces several unexpected risk or errors. External calls may execute malicious code in that contract. So every external call should always treat as potential security. When it is not possible or undesirable to remove external calls, use recommendations.

Problem: Malicious contract able to attack smart contract by using Reentrancy.
Main contract:
This contract enables the users can deposit and withdraw their funds either internally or external from contract.

```
function deposit (address to) external payable {
   _balanceof [to] += msg.value;
 }
```

Fig: deposit() function

```
function withdrawEquity () external payable {
uint x = _balanceof [msg.sender];
msg.sender.call.value(x) ();
// vulnerability due to state variable updated after call()
_balanceof [msg.sender] = 0;
```

Fig: **withdraw()** function

Algorithm1: Reentrance Main Contract
Any user can deposit ether into Reentrance main contract through their address and update balance
Input:
$Uaddr_1$ ← User Address1
U_{bal1} ← update balance Uaddr1
D_{usr1} ← deposit into $Uaddr_1$
Output: update the balance of the user
Begin: $Uaddr_1$ ← Fetch user address1
1. Call **deposit ()** function
2. D_{usr1} ← deposit 30 Ether in to main contract
3. U_{bal1} ← update balance of user1 as 30 ETH.
4. **End**

Attacker Contract:
Attacker deployed malicious contract and he will deposit 10 ETHER in to main contract by using malicious contract address and able to withdraw 40 ETHER from main contract by using RECURSIVE call function i.e **fallback()** function.

```
function () payable external{
vul.withdrawEquity ();
}
```

Fig: Recursive **fallback** () function

```
function winnerWinnerChickenDinner () public {
_owner.transfer (this.balance);
}
```

Fig: Attacker Transfer amount To his account

Algorith2: **Malicious contract**
Input:
$Uaddr_2$ ← Attacker Address
$Uaddr_3$ ← Malicious contract Address
U_{bal3} ← Balance of Malicious contract address
D_{usr32} ← Deposit in to malicious contract address
1. Begin: $Uaddr_2$ ← Fetch attacker address
2. $Uaddr_3$ ← Fetch Malicious contract address
Call **deposit()** function
3. Dusr3 ← deposit 10 ETHER into main contract through malicious contract address.
5. Ubal3 ← update balance of malicious contract address.
6. Attacker call Recursive **Fallback ()** function inside malicious contract.
7.**withdrawEquity()** function called recursively inside fallback() function until 40 Ether drain from main contract
8. In **withdrawEquity ()** function there is a vulnerability, that the state variable of the current contract is not updated after withdrawn amount from the state variable.
9.So immediately the control transferred to the calling contract, before finishing the first invocation in the main contract, due to characteristic of msg.sender.call.value(amount)();
10. And the withdrawn () function is called recursively until the all the amount drain from the main contract or run out of gas.
11.attacker called **winnerWinnerChickenDinner ()** function to transfer amount in to his account

3. Observations and Results

We have performed experimental observations on JVM, web3 provider using Ganache and Injected web3 using Rinkeby Test Network. We have used solidity v0.6.1 language using REMIX IDE.And the Ether value in INR 24,144.32.

Table 1: Experimental results obtained after deployed on Rinkeby test network using metalmark for dao attack

Sno	Operation performed	Input to the function call	Updated balance of the user (ether)	Update balance of the attacker (ether)	Transaction cost (gas)	Transaction fee (ETHER)	Total cost in Rupees (INR)
1	Contract deployed	-	-	-	197221	0.014791575 Ether	314.2490
2	Donation to the user account with address	20 Ether	20Ether	-	42515	0.003188625 Ether	67.74277
3	Attacker Deploy malicious contract	-	-	-	270340	0.01946448 Ether	413.5255
4	Attacker Donation to the malicious contract address	10 ether	-	10 ether stored in Main contract	42515	0.003358685 Ether	71.3557
5	Attacker called fallback function	-	-	30 Ether Received by the Malicious contract	317598	0.00853875 Ether	181.4069
7	Attacke called jackpot function	-	-	30 Ether Transferred to attacker account	32541	0.00242288 Ether	51.4727

Table 2: Experimental results obtained after deployed on Rinkeby test network using metamask for dao attack

S. No.	Operation performed	transaction time	Number of blocks mined	Number of transactions mined	Time taken to mine	Amount of information mined (in bytes)
1.	Contract deployed	38.31seC	6872401	58	15sec	10,085 bytes
2.	Donation to the user account with address	35.23 SEC	6872503	5	15 sec	1,641 bytes
3.	Attacker Deploy malicious contract	37.46sec	6872655	6	15 sec	2,874 bytes
4.	Attacker Donation to the malicious contract address	38.55 sec	6872713	7	15 sec	1,824 bytes
5	Attacker called fallback function	33.67 sec	6872976	66	15 sec	9,788 bytes
6	Attacke called jackpot function	30.35sec	6873050	9	15 sec	2,674 bytes

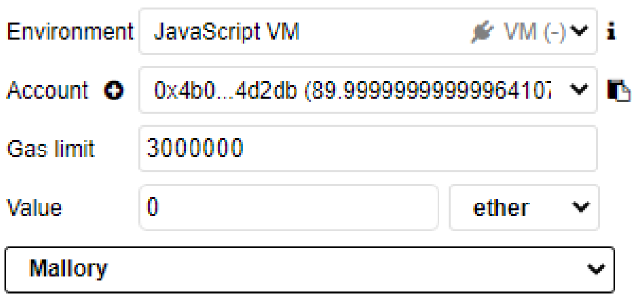

Figure 1: Attacker deposited 10 ether in to malicious contract using JVM Environment

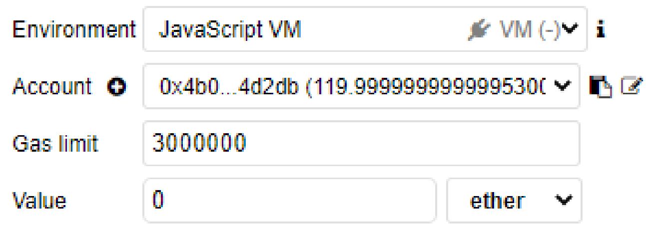

Figure 2: Attacker Gained 30 Ether by depositing 10 ETHER into Malicious contract using JVM Environment

Table 3: Experimental results obtained after deploy o Rinkeby test network using metalmark for underflow attack (bank balance transfer from sender to receiver)

Sno	Operation performed	Contribution to the sender	Case1:transfer1		Case 2:transfer2		Case 3:transfer3		Transaction cost (gas)	Transaction fee (ETHER)	Total cost in Rupees (INR)
			input	output	input	output	input	output			
1	Contract deployed for bank customer	-	-	-		-	-	-	192793	0.016194612	38.64
2	Contribution to the sender	5							41308	0.003387256	80.81
3	Sender balance	-	5	3	5	0	5	Under flow attack and balance changed to maximum value	Cas1; 48809	0.004197574	100.154
4	Value to be transfer	-	2	-	5		6	-	Cas2; 48809	0.005197594	124.01
5	Receiver balance	-	0	2	0	5	0	6	Cas3; 48809	0.003198572 -	76.31

Table 4: Experimental results obtained after deploy on Rinkeby test network using metamask for Reentrancy attack (bank balance transfer from sender to receiver

Sno	Operation performed	Input to the function call	Updated balance of the user (ether)	Update balance of the attacker (ether)	Transaction cost (gas)	Transaction fee (ETHER)	Total cost in Rupees (INR)
1	Main Contract deployed	-	-	-	195493	0.019158314 Ether	457.12
2	Donation to the user account address	30 Ether	30 Ether	-	42515	0.004294015 Ether	102.456
3	Attacker Deploy malicious contract	-	-	-	270340	0.02757468 Ether	657.94
4	Attacker Donated to malicious contract address	10 Ether	-	Malicious contract address=10 ether	42515	0.00433653 Ether	103.47
5	Attacker called recursive Fallback Function to receive ETHER	-	-	40 ether received by Malicious contract		0.01345891 Ether	321.134
6	Attacker called Winner function to received Ether to his account	-	-	40 Ether Received by Attacker by depositing 10 ETH to Malicious contract	23586	0.002500116 Ether	59.653

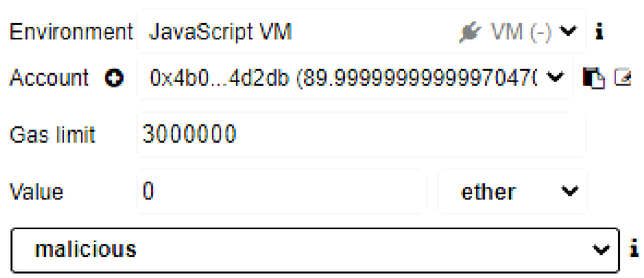

Figure 3: After donating 10 Ether in to malicious contract

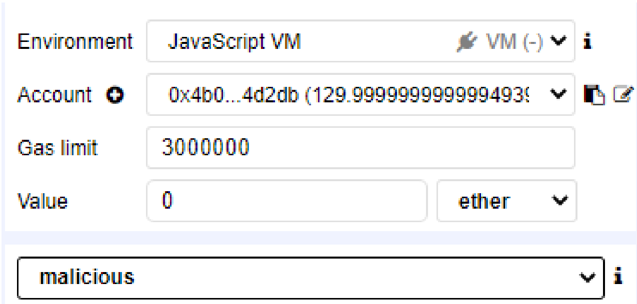

Figure 4: Attacker gained 40 Ether using Reentrancy Attack

Table 5: Security Flaw

S.no	Type of attack	Security Flaw
1	DAO	1 addr.call.value (amount) (“”); when using this instruction in the contract it will forward the all remaining gas and open up the recipient to perform some complex operations. It includes calling back to the sending contract or other states may change, that may not we expect. As a result attackers attack the main contract and drain the whole amount by using fallback () function.
2	Underflow	**1. Overflow:** it occurs when uint8(255)+uint8(1)==0. It occurs when the operation performed; the value stored in a fixed variable is outside the range of the variable type. Attacker try to make the balance is zero value. **2. UNDERFLOW:** An under flow occurs when operation performed is uint8(0)-uint8(1)=255. Attacker take advantage by using this operation.
3	Reentrancy	1. Ether Transfer can always include code execution, So the recipient could be a contract that calls back into **withdraw** function. This would let get multiple refunds and basically retrieve all the ether in the contract.By using **call** instruction, it will always forward remaining gas to recepent contract.. 2. Attacker Using RECURSIVE **fallback ()** function in external contract

Table 6: Secure Method

S. No	Type of attack	SECURE METHODS
1	DAO	1.**First Perform the following checks** Who called the function, are the arguments in given range, did they send enough Ethers, does the person have enough tokens. 2. If all the above checks are passed, then effects to the state variables of the current contract should made next. And the interaction with other contracts should be the last step in the any function i.e msg.sender.call.value(amount(‘’); 3. Include some kind of **fail – safe mechanism** to check all the above conditions. If it fails, the contract automatically switches in to some kind of “**fail safe** “mode. Which Disables most of the features, hands over control to a fixed or trusted third party or just convert the contract in to a simple “give me back my money” contract. 4. Include function **modifier** to check the condition before executing the function.
2	Underflow	1.Use **require** condition to limit the size of inputs to a reasonable range and Ex: **require (balanceof [to] +_value)>=balance of [to}.** 2. use **safeMath** library. 3.Use the **SMT** checker.
3	Reentrancy	1. If you are making a call to an untrusted external contract, avoid state changes after the call. 2.Use check effect interaction pattern as discussed in above DAO from step 1 to 4.

4. Suggestions and Recommendations

After practically analyzing all the three smart contract vulnerabilities, we conclude that while writing smart contracts using solidity on Ethereum blockchain. It is compulsory to check the conditions like whether sufficient ethers are available before transfer, is the receipt is trusted one, is the integers within the limitation after exchange of ethers, check whether the current contract state variables are updated before the ether transfer takes place. And we are going to check all these conditions on our project, supply chain management for textile industry as a future scope for this paper.

References

[1] Security Vulnerabilities in Ethereum Smart Contracts, Markus,Flatscher, iiWAS ’18,November 19-21, 2018,Yogyakarta, Indonesia 2018, ISBN 78-1-4503-6479-9/18/11,https://doi.org/10.1145/3282373.3282419.
[2] SMT – based Verification of Solidity smart Contracts, Leonardo Alt and Christian Reitwiessner Ethereum Foundation
[3] Defects and Vulnerabilities in Smart Contracts, a Classification using the NIST Framework, Wesley Dingman,Aviel Cohen,Nick Ferrara,Adam Lynch,Patrick Jasinski,Paul E.Black,Lin Deng.
[4] https://swcregistry.io/docs/swc-107
[5] https://consensys.github.io/smart-contract-best-

practices/known_attacks/#reentrancy

[6] Supply chain digitization trends: An integration of knowledge management Dara.Schniederjansa,Carla Curadob, Mehrnaz,,Khalajhedayatia, https://doi.org/10.1016/j.ijpe.2019.7.012

[7] The Evolution of Blockchain: A Bibliometric Study MOHAMMAD DABBAGH 1,MEHDI SOOKHAK2, AND NADER SOHRABI SAFA3.

[8] Security Vulnerabilities in Ethereum smart Contracts, Alexander Mense, Markus Flatsscher,2018,November 19-21.2018,Yogyakarta,Indonesia,2018,https://doi.org/10.1145/3282373.3282419

[9] KEVM : A Complete Formal Semantics of the Ethereum Virtual Machine, Everett Hildenbrandtac, Manasvi Saxena, Nishant Rodriguesb, Xiaoran Zhuae,2018 IEEE 31st Computer Security Foundations Symposium

[10] **Security considerations** https://solidity.readthedocs.io/en/v0.6.10/securityc-onsiderations.html?highlight=security%20considerations.

[11] Satchain: Secured Autonomous Transactions in Supplychain Using: Block chain, B.Ratnakanth, K.VenkataRamana, IJITEE, ISSN:2278-3075, Volume-9 Issue-6, April 2020.

[12] Systematic Approach To Analyze Attacks on SCM: Using Blockchain, B.Ratnakanth, Dr.K.VenkataRamana, IJRTE, ISSN:2277-3878,Volume-9 Issue-2,July2020

[13] Secure payment system in supplychain management using blockchain technologies, www.jetir.org, ISSN-2349-5162,© 2019 JETIR June 2019, Volume 6, Issue

[14] A Survey of Attacks on Ethereum Smart Contracts (SOK), Nicola Atzei, Massimo Bartoletti (B), and Tiziana Cimoli, Springer –Verlag GmbH Germany 2017 M.Maffei and M.Ryan (Eds.), DOI:10.1007/978-3-662-54455-68.