Efficient Methods to Avoid Smart Contract Vulnerabilities Using Block Chain

B. Ratnakanth¹, M. Sahiti², Dr. K. Venkata Ramana³

¹Vignan Institute of Technology & Science, Hyderabad, A.P., India
²Dep: Computer Science and System engineering, Andhra University, Visakhapatnam, A.P., India
³Assistant 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. Schniederjanssa, Carla Curadob, Mehrnaz Khalajghedayatia 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 saf3]. The blockchain 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() ("w"). 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 30th 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 Reentrancy 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) ("w");

```solidity
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:**
Uaddr₁ ← User Address1
Ubal₁ ← update balance Uaddr1
Dusr₁ ← deposit into Uaddr₁

**Output:** update the balance of the user
1. Begin: Uaddr₁ ← Fetch user address1
2. Call deposit() function
3. Dusr₁ ← deposit 20 ether into DAO
4. Ubal₁ ← update balance of user1

```solidity
function withdraw (uint amount) external {
    if (credit [msg.sender]=amount)
    msg.sender.call.value (amount);
    credit [msg.sender]=amount;
}
```

**Fig:** DAO contract withdraw function

Algorithm2: ATTACKER CONTRACT

Attacker can deposit ether into DAO. And attack the DAO by Recursive fallback () function.

**Input:**
Uaddr₂ ← Attacker Address2
Uaddr₃ ← Attacker contract Address3
Ubal₃ ← Attacker contract balance Uaddr₃
Dusr₃₂ ← deposit into Uaddr₃

1 Begin: Uaddr₂ ← Fetch attacker address2
2. Uaddr₃ ← Fetch attacker contract address3
3. Call deposit() function
4. Dusr₃ ← Attacker deposited 10 Ether into DAO through Malicious contract address
5. Ubal₃ ← 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 uint256 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

```solidity
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 condition 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:
Uaddr1 ← sender account
Daddr1 ← sender account
V_d ← value to be transfer
Daddr2 ← receiver account
B_bal1 ← Balance of the sender account
B_bal2 ← receiver account

Output:
1. Begin:
2. Fetch sender account ← Uaddr1
3. Contribute to the sender account contribute() function
4. V_d ← 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/overflow 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.

```solidity
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:
Uaddr1 ← User Address1
Ubal1 ← update balance Uaddr1
Dusr1 ← deposit into Uaddr1
Output: update the balance of the user

Begin: Uaddr1 ← Fetch user address1
1. Call deposit () function
2. Dusr1 ← deposit 30 Ether in to main contract
3. Ubal1 ← 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.

```solidity
function () payable external{  
    vul.withdrawEquity ();  
}
Fig: Recursive fallback () function
```
6. 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.

7. 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)();`.

8. And the `withdraw()` function is called recursively until the all the amount drain from the main contract or run out of gas.

9. 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.

<p>| Table 1: Experimental results obtained after deployed on Rinkeby test network using metalmark for dao attack |
|-----------------------------------------|-----------------------------------------|-----------------------------------------|-----------------------------------------|-----------------------------------------|-----------------------------------------|</p>
<table>
<thead>
<tr>
<th>Sno</th>
<th>Operation performed</th>
<th>Input to the function call</th>
<th>Updated balance of the user(ether)</th>
<th>Update balance of the attacker (ether)</th>
<th>Transaction cost (gas)</th>
<th>Transaction fee (ETHER)</th>
<th>Total cost in Rupees (INR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Contract deployed</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>197221</td>
<td>0.014791575 Ether</td>
<td>314.2490</td>
</tr>
<tr>
<td>2</td>
<td>Donation to the user account with address</td>
<td>20 Ether</td>
<td>20Ether</td>
<td>-</td>
<td>42515</td>
<td>0.003188625 Ether</td>
<td>67.74277</td>
</tr>
<tr>
<td>3</td>
<td>Attacker Deploy malicious contract</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>270340</td>
<td>0.01946448 Ether</td>
<td>413.5255</td>
</tr>
<tr>
<td>4</td>
<td>Attacker Donation to the malicious contract address</td>
<td>10 ether</td>
<td>-</td>
<td>10 ether stored in Main contract</td>
<td>42515</td>
<td>0.003358685 Ether</td>
<td>71.3557</td>
</tr>
<tr>
<td>5</td>
<td>Attacker called fallback function</td>
<td>-</td>
<td>-</td>
<td>30 Ether Received by the Malicious contract</td>
<td>317598</td>
<td>0.00853875 Ether</td>
<td>181.4069</td>
</tr>
<tr>
<td>7</td>
<td>Attacker called jackpot function</td>
<td>-</td>
<td>-</td>
<td>30 Ether Transferred to attacker account</td>
<td>32541</td>
<td>0.00242288 Ether</td>
<td>51.4727</td>
</tr>
</tbody>
</table>

<p>| Table 2: Experimental results obtained after deployed on Rinkeby test network using metamask for dao attack |
|-----------------------------------------|-----------------------------------------|-----------------------------------------|-----------------------------------------|-----------------------------------------|-----------------------------------------|</p>
<table>
<thead>
<tr>
<th>S. No</th>
<th>Operation performed</th>
<th>transaction time</th>
<th>Number of blocks mined</th>
<th>Number of transactions mined</th>
<th>Time taken to mine</th>
<th>Amount of information mined (in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Contract deployed</td>
<td>38.31sec</td>
<td>6872401</td>
<td>58</td>
<td>15sec</td>
<td>10,085 bytes</td>
</tr>
<tr>
<td>2.</td>
<td>Donation to the user account with address</td>
<td>35.23 SEC</td>
<td>6872503</td>
<td>5</td>
<td>15 sec</td>
<td>1,641 bytes</td>
</tr>
<tr>
<td>3.</td>
<td>Attacker Deploy malicious contract</td>
<td>37.46sec</td>
<td>6872655</td>
<td>6</td>
<td>15 sec</td>
<td>2,874 bytes</td>
</tr>
<tr>
<td>4.</td>
<td>Attacker Donation to the malicious contract address</td>
<td>38.55 sec</td>
<td>6872713</td>
<td>7</td>
<td>15 sec</td>
<td>1,824 bytes</td>
</tr>
<tr>
<td>5.</td>
<td>Attacker called fallback function</td>
<td>33.67 sec</td>
<td>6872976</td>
<td>66</td>
<td>15 sec</td>
<td>9,788 bytes</td>
</tr>
<tr>
<td>6.</td>
<td>Attacker called jackpot function</td>
<td>30.35 sec</td>
<td>6873050</td>
<td>9</td>
<td>15 sec</td>
<td>2,674 bytes</td>
</tr>
</tbody>
</table>

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 on Rinkeby test network using metalmark for underflow attack (bank balance transfer from sender to receiver)

<table>
<thead>
<tr>
<th>Sno</th>
<th>Operation performed</th>
<th>Contribution to the sender</th>
<th>Case 1:transfer1</th>
<th>Case 2:transfer2</th>
<th>Case 3:transfer3</th>
<th>Transaction cost (gas)</th>
<th>Transaction fee (ETHER)</th>
<th>Total cost in Rupees (INR)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Contract deployed for bank customer</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>192793</td>
<td>0.016194612</td>
<td>38.64</td>
</tr>
<tr>
<td>2</td>
<td>Contribution to the sender</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>41308</td>
<td>0.003387256</td>
<td>80.81</td>
</tr>
<tr>
<td>3</td>
<td>Sender balance</td>
<td>-</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>16103</td>
<td>0.004197594</td>
<td>124.01</td>
</tr>
<tr>
<td>4</td>
<td>Value to be transfer</td>
<td>-</td>
<td>2</td>
<td>-</td>
<td>5</td>
<td>5</td>
<td>0.005197594</td>
<td>76.31</td>
</tr>
<tr>
<td>5</td>
<td>Receiver balance</td>
<td>-</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0.003198572</td>
<td>67.31</td>
</tr>
</tbody>
</table>

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

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

Figure 3: After donating 10 Ether in to malicious contract
Figure 4: Attacker gained 40 Ether using Reentrancy Attack

Table 5: Security Flaw

<table>
<thead>
<tr>
<th>S.no</th>
<th>Type of attack</th>
<th>Security Flaw</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DAO</td>
<td>1. addr.call.value (amount) (&quot;&quot;&quot;); when using this instruction in the contract it will forward 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.</td>
</tr>
<tr>
<td>2</td>
<td>Underflow</td>
<td>1. <strong>Overflow</strong>: 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. <strong>UNDERFLOW</strong>: An underflow occurs when operation performed is uint8(0)-uint8(1)==255. Attacker take advantage by using this operation.</td>
</tr>
<tr>
<td>3</td>
<td>Reentrancy</td>
<td>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 recepient contract.. 2. Attacker Using RECURSIVE fallback () function in external contract</td>
</tr>
</tbody>
</table>

Table 6: Secure Method

<table>
<thead>
<tr>
<th>S. No</th>
<th>Type of attack</th>
<th>SECURE METHODS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DAO</td>
<td>1. <strong>First Perform the following checks</strong>  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(&quot;&quot;&quot;)); 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 &quot;fail safe&quot; 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 &quot;give me back my money&quot; contract. 4. Include function modifier to check the condition before executing the function.</td>
</tr>
<tr>
<td>2</td>
<td>Underflow</td>
<td>1.Use require condition to limit the size of inputs to a reasonable range and Ex: require (balanceof [to] +_value)&gt;=balance of [to]. 2. use safeMath library. 3.Use the SMT checker.</td>
</tr>
<tr>
<td>3</td>
<td>Reentrancy</td>
<td>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.</td>
</tr>
</tbody>
</table>

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


practices/known_attacks/#reentrancy


[12] Systematic Approach To Analyze Attacks on SCM: Using Blockchain, B.Ratnakanth, Dr.K.Venkataramana, IJRTE, ISSN:2277-3878, Volume-9 Issue-2, July 2020


[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.