Erzincan Üniversitesi
Fen Bilimleri Enstitüsü Dergisi
2023, 16(1), 196-211
ISSN: 1307-9085, e-ISSN: 2149-4584
Araştırma Makalesi

Erzincan University
Journal of Science and Technology
2023, 16(1), 196-211
DOI: 10.18185/erzifbed.1105551
Research Article

Smart Contract Security Vulnerabilities

Ruhi Taş[1*]

[1] Turkish Radio Television Co.

Received:18/04/2022, **Revised:** 14/09/2022, **Accepted:** 15/09/2022, **Published:** 30/03/2023

Abstract

A smart contract is a concept of computer protocols that helps to facilitate blockchain technology. This blockchain-based smart contract is a public ledger of all participating transactions. It is considered a self-executable application and contains predetermined rules. It also operates by decentralizing networks that are shared between all parties, and this execution of contracts between parties could be securely done without a middleman or a third party. With blockchain technology, developers could provide an efficient framework and ensure security issues. While the new blockchain has successfully been developed to prevent the problems of fraud and hacking, there is still a considerable risk concerning security and confidentiality. Therefore, we should not underestimate this matter. This study aims to review the potential risks that may take place on blockchain-based smart contracts. In addition, the options that may assist application developers in order to provide viable guidance, and to avoiding these security vulnerabilities.

Keywords: Blockchain, Smart Contract, Vulnerability, Ethereum, Solidity

Akıllı Sözleşme Güvenlik Zaafiyetleri

Öz

Akıllı sözleşme, blok zinciri teknolojisini kolaylaştırmaya yardımcı olan bir bilgisayar protokolleri kavramıdır. Bu blok zinciri tabanlı akıllı sözleşme, katılan tüm işlemlerin halka açık bir defteridir. Kendi kendine çalıştırılabilir bir uygulama olarak kabul edilir ve önceden belirlenmiş kurallar içerir. Ayrıca, tüm taraflar arasında paylaşılan ağları merkezi olmayan hale getirerek çalışır ve taraflar arasındaki bu sözleşmelerin yürütülmesi, bir aracı veya üçüncü bir taraf olmadan güvenli bir şekilde yapılabilir. Blockchain teknolojisi ile geliştiriciler verimli bir çerçeve sağlayabilir ve güvenlik sorunlarını sağlayabilir. Yeni blok zinciri, dolandırıcılık ve bilgisayar korsanlığı sorunlarını önlemek için başarıyla geliştirilmiş olsa da, güvenlik ve gizlilik konusunda hala önemli bir risk var. Bu nedenle bu konuyu hafife almamalıyız. Bu çalışma, blockchain tabanlı akıllı sözleşmelerde yer alabilecek potansiyel risklerin incelenmesi amaçlamaktadır. Ayrıca geliştiricilere rehberlik ederek, olası güvenlik açıklarından kaçınmak için uygulama geliştiricilere yardımcı olunması sağlanmıştır.

Anahtar Kelimeler: Blok zinciri, Akıllı sözleşmeler, Zaafiyet, Ethereum, Solidity

***Corresponding Author**: ruhi.tas@trt.net.tr
Ruhi Taş, https://orcid.org/0000-0002-7741-7715

1. Introduction

Satoshi Nakamoto proposed a system with several rules to build the blockchain infrastructure in 2008. In particular, he created the first cryptocurrency, a form of digital money based on cryptography, which prioritizes the level of security [1]. In 2009, Bitcoin cryptocurrency was created and started to be traded on the market. It gained popularity as a result of the use of the Bitcoin blockchain infrastructure. Bitcoin is the first application that managed to disable central control [2]. Blockchain, which uses a decentralized structure, offers a reliable system thanks to its immutability and distributed record structure. Particularly, participants who do not know each other confirm the accuracy of the transactions within the framework of certain rules and keep them [3]. The blockchain provides an unalterable permanent record of transactions on a network. The system uses a decentralized ledger similar to a database. In this system, the people involved in the system can see the records of all transactions, if they wish. This feature causes it to differentiate from traditional databases [4]. In the following process, their usability in alternative areas has emerged. It has spread to a wide area such as supply chain [5], agricultural practices, insurance [6], health [7] to secure digital rights management [8], pharmaceuticals [9], financial transactions, and trade and commerce [10].

A smart contract is a leading product of Ethereum blockchain; and it is one of the most useful cryptocurrencies. Ethereum [11] allows other blockchain applications to be built on it [12]. With the use of smart contracts in blockchain systems, special applications have started to be developed within contract-based privatized sectors [13]. It is stated that blockchain technology provides security and a stable working environment, especially in online transactions. However, the lack of standards and weaknesses in application development can pose serious risks. These vulnerabilities could take place at different levels: the blockchain framework level, the peer-to-peer network level, and the blockchain application at the reasonable contract level. In this study, we primarily focus on security matters and how to prevent these vulnerabilities which may take place on the blockchain-based smart contracts.

2. What is Smart Contract?

A smart contract is code developed as a script that is pinned to a blockchain or similar distributed infrastructure. Triggered by the blockchain transaction and verified over the network, it is used to execute predefined actions. Since the terms of a smart contract are transparently stored on the blockchain, it can always be checked whether all parties are working as intended. In this way, trust problems between the related parties are reduced. Smart contracts can be written as software scripts, just like scripts running in non-blockchain applications. The term smart contract and the underlying idea predates the emergence of Bitcoin. Szabo [14] defined the smart contract as a piece of computerized transaction. The protocol, which meets the terms of payment, contract terms such as confidentiality or execution, enables the realization of transactions in accordance with the criteria by making the necessary transactions. Solidity, Ethereum The design of such systems has legal, economic and technical bases. Therefore, smart contracts require interdisciplinary analysis.

3. Potential Risks of Smart Contract Implementations on Blockchain

The Smart Contract definition was first proposed by Nick Szabo in 1994 [14]. According to the structure defined by Szabo, it has been stated that the contracts that have to be used in some cases can be converted into codes that computers can process. In this way, it will be able to be stored and copied in the system. When applied to the blockchain, it can be activated as a control mechanism in the nodes on the network [15]. Smart contracts are the writing of a contract on the lines of code, and the transactions are executed according to the terms of this contract [16]. Contracts developed to provide the necessary controls are uploaded to the nodes. Later, other nodes in the blockchain will be enabled to communicate with the same mechanism. Smart contracts ensure that transactions between nodes are carried out securely.

Smart contract programming requires different features from the standard application development methods. If there is any failure of the applications, the cost for defective software could be high. The expected changes could also be difficult, which can be compared to a hardware design and programming. It is usually written in a simple language of smart contract languages, expressions, operators, functions, and variables. Although they are already quite abstract and difficult to understand, the components of a smart contract can be expressed in part by name from anywhere in the program; sometimes it is almost impossible to see how different parts interact and fit.

Smart contracts are self-distributed computer programs which are executed on the blockchain framework. Popular applications of smart contracts include cryptocurrencies and online gambling; those applications often involve financial transactions which consider as a part of the contract. Similar to conventional programs, smart contracts are written in Solidity [9], and can contain security vulnerabilities which could potentially lead to attacks. Unlike ordinary programs, the problem is overcome by an inability to correct smart contracts.

In recent years, some studies have demonstrated that all blockchain-based smart contracts contain some security vulnerabilities [17] and could encounter the attacks which can potentially lead to devastating losses [18, 19]. For example, structure attacks; such as forks [20], DDoS attacks [18], majority attacks [21], and double spending [22]. There is an increasing number of occurrences in smart contracts, and financial losses have been reported [18]. These attacks have been found to exploit the vulnerabilities in smart contracts. To give an example, there is an incident of a DAO attack, and this is considered as a result of a minor error in the DAO contract [23]. Moreover, on June 17, 2016, there were over 3.5 million of Ether have been stolen from the DAO smart contract [24].

According to a report published in 2018, the smart contract contains approximately 4.4 million dollars with an Ethereum value at that time, this could be a potential vulnerability for hackers to exploit it [25, 26]. Currently, 4.008 DApps and 7.16K smart contracts are executing on the blockchain networks (Figue 1) [27]. The most widespread blockchain networks have several security concerns that could be exploited by hackers.

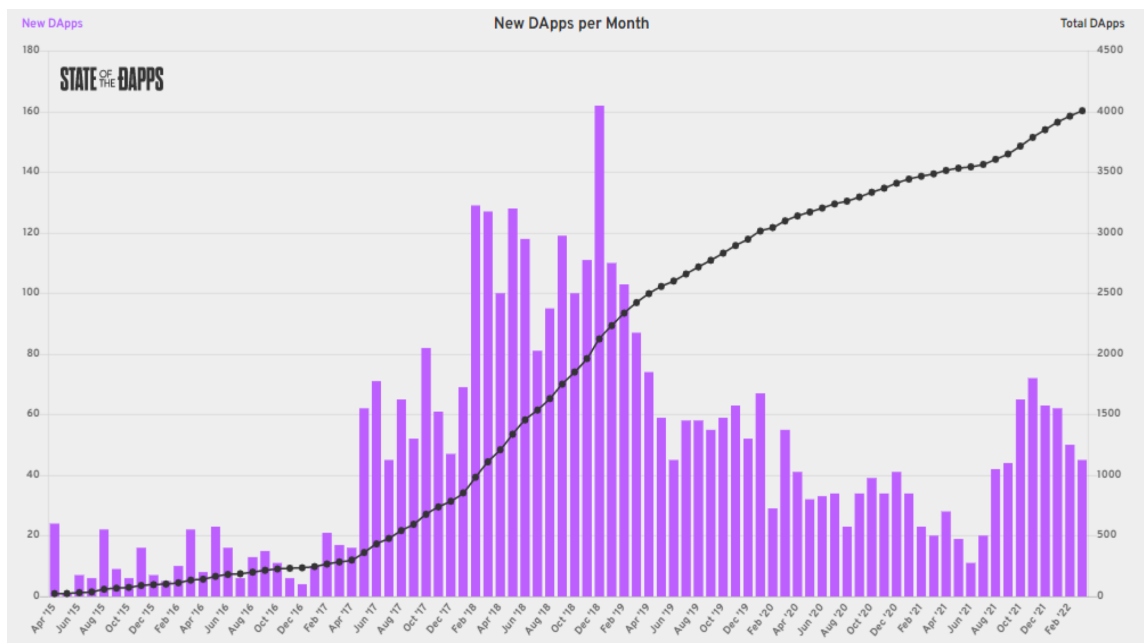

Figure 1. DApps per Month [27].

4. Material and Methods

In order to obtain the articles on this topic, we first establish a possibility of the following search criterias: search queries “smart contract vulnerabilities”, “contract vulnerabilities”, “smart contract attacks” applied to Arxiv [28] (143 article), and IEEExplore [29] (443 article) digital library and general search. We will then obtain search results by reviewing title and abstracts. Non-English articles and those with different topics will also be eliminated.

5. Conclusions

As a result of the examination of the articles that meet the criteria, vulnerabilities in smart contracts that can easily be mistaken have been identified.

Reentrancy Attacks

Reentrancy attack on the smart contracts is a well-known vulnerability because this kind of attack allows attackers to take control of the flow on smart contracts. It is also known as an “unknown call,” or a “recursive call” vulnerability. Multiple parallel external initializations are possible using the structures call family. If the global state is not managed suitably, a contract may be vulnerable to delay attacks [30]. Hackers can take over the control flow, and make changes to your data that the calling function is not expected [31]. When the linked cross-function is in a racing state, two different functions will then operate on the same global state [30]. Smart contracts have found the opportunity to be used with Blockchain 2.0. In particular, Ethereum has started to be used in the infrastructure of crypto money. The particularly risky situation is when one contract calls another, the current execution has to wait for the call to finish. This can cause a problem especially when the recipient of the call is using the intermediate state the caller is in [32, 33]. An attacker can perform a parallel attack by using two different functions that share at the same state [34]. It is advised to use a built-in transfer() function, specifically when we make external calls to another untrusted contracts.

We also need to confirm all logics that could change the state variables that could happen before it sent out of the contract [35].

A code written in this way can be easily exploited by an attacker. Line 15 contains a bypassable vulnerability that can be recalled by the attacker (Figure 2).

```
1   contract Bank {
2       uint256 public paraCekmeLimiti = 1 ether;
3       mapping(address => uint256) public sonParaCekmeZamani;
4       mapping(address => uint256) public kalanPara;
5
6       function mevduat() public payable {
7           mevduatlar[msg.sender] += msg.value;
8       }
9
10      function paraCekmeFonu (uint256 _miktar) public {
11          require(mevduatlar[msg.sender] >= _miktar);
12          require(_miktar <= paraCekmeLimiti);
13          require(now >= sonParaCekmeZamani[msg.sender] + 1 weeks);
14          require(msg.sender.call.value(_miktar)());
15          mevduatlar[msg.sender] -= _miktar;
16          sonParaCekmeZamani[msg.sender] = now;
17      }
18  }
19
```

Figure 2. Reentry sample code

```
24  function Attacker() public payable {
25
26          require(msg.value >= 1 ether);
27          // send eth to the mevduat() function
28          Bank.mevduat.value(1 ether)();
29          // start the attack
30          Bank.paraCekmeFonu(1 ether);
31    }
32   function () payable {
33          if (etherStore.balance > 1 ether) {
34              bank.paraCekmeFonu(1 ether);
35          }
36    }
```

Figure 3. Reentry Attacker sample code

In cases where the necessary control is not performed by calling the same function again on the 30th and 34th lines, the transactions in the other expenditure can be started before the expenditure transaction is concluded (Figure 3).

Gas Limit Attacks

This attack is possible in a contract that accepts basic data and uses it to make another contract through the low-level address.call () function, as is often the case in multi-signed and transactional situations [31]. It is claimed that we could avoid this potential vulnerability by looping over arrays of unknown length, setting an upper limit for the array length, and controlling the loop by inspecting the gas limit [36].

```
1   struct Payment {
2       address paymentAddress;
3       uint256 value;
4   }
5
6   Payment[] payments;
7   uint256 nextPaymentI;
8
9   function paymentOut() {
10      uint256 i = nextPaymentI;
11      while (i < payments.length && msg.gas > 200000) {
12        payments[i].paymentAddress.send(payments[i].value);
13        i++;
14      }
15      nextPaymentI = i;
16  }
```

Figure 4. Gas Limit sample code

It is necessary to ensure that there are no undesirable consequences if other transactions are processed while waiting for the next iteration of the PaymentOut() function. Therefore, this pattern should only be used if absolutely necessary (Figure 4).

Arithmetic Issues (Integer Overflow)

Integer overflow is a type of errors that can be found in many programming languages. It could cause a serious security vulnerability in blockchain applications. For example, if a loop counter overflow and creates an infinite loop, the contract then can be completely frozen. This overflow can be used by an attacker, especially when there is an increasing number of iterations in the loop which has been registered by new users of the agreement [30, 37]. Therefore, when writing contracts, we need to utilize a secure math libraries for all arithmetic operations, such as OpenZeppelin’s SafeMath library [38].

Even the “solidity” library used in the example below has a great influence on whether the system malfunctions or not. When version 0.8.0 is used, the system works properly as a result of compilation (Figure 5, Figure 6).

```
1  pragma solidity 0.7.0;
2
3  contract ChangeBalance {
4      uint8 public balance;
5
6      function decrease() public {
7          balance--;
8      }
9
10     function increase() public {
11         balance++;
12     }
13 }
```

```
1  pragma solidity 0.8.0;
2
3  contract ChangeBalance {
4      uint8 public balance;
5
6      function decrease() public {
7          balance--;
8      }
9
10     function increase() public {
11         balance++;
12     }
13 }
```

Figure 5. Overflow sample code

```
[vm] from: 0x5B3...eddC4 to: ChangeBalance.decrease() 0x7EF...8CB47 value: 0 wei data: 0xd73...2d955 logs: 0 hash: 0xa14...7259f

transact to ChangeBalance.decrease errored: VM error: revert.

revert
        The transaction has been reverted to the initial state.
Note: The called function should be payable if you send value and the value you send should be less than your current balance.
Debug the transaction to get more information.
```

Figure 6. Overflow compile error

Delegatecall

A message except that the calls DELEGATECALL offered under the contract is the same as the destination address in the code calls. The destination address in the code call contract (Jiang et al., 2018) is carried out under a separate message call from the same DELEGATECALL called the message there are certain types of calls. When using DELEGATECALL, both the library contract and the possible negotiating conditions of the interview contract should be considered [30].

```
Delegatecall.sol
1  pragma solidity ^0.8.10;
2
3  contract Lib {
4      address public owner;
5
6      function pwn() public {
7          owner = msg.sender;
8      }
9  }
10
11 contract HackMe {
12     address public owner;
13     Lib public lib;
14
15     constructor(Lib _lib) {
16         owner = msg.sender;
17         lib = Lib(_lib);
18     }
19
20     fallback() external payable {
21         address(lib).delegatecall(msg.data);
22     }
23 }
24
25 contract Attack {
26     address public hackMe;
27
28     constructor(address _hackMe) {
29         hackMe = _hackMe;
30     }
31
32     function attack() public {
33         hackMe.call(abi.encodeWithSignature("pwn()"));
34     }
35 }
```

Figure 7. Delegatecall sample code

At first, HackMe works according to whether the pwn function is present in the Contract (Figure 7). Since there is no pwn function, HackMe's fallback function is triggered, which calls the Lib contract with the pwn function's signature. That the lib contract has the pwn function definition and the owner is set to msg.sender. With context protection msg.sender can now be used as owner of HackMe contract.

State Variable Visibility

Visibility is used to determine whether the functions should be invoked by users internally or externally from different contracts [30]. Variables or functions can be defined as public, private, or internal [27]. Private variables can only be accessed by a declaring contract itself, and internal variables can be accessed by defining contracts and the contracts derived from them [38]. It is obligatory to determine a visibility of all functions that are used in a contract in a controlled manner.

It is important to define public, internal and private accessibility in the example so that it does not cause clarity (Figure 8).

```
16  contract Accessibility {
17    uint public publicInt; //public
18    uint internal internalInt; //internal
19    uint private privateInt; //private
20  }
21  // can get and set from other smart contracts
22  contract AccessPublic {
23    Accessibility accessibility = new Accessibility();
24    function accessPublicInt() public view returns(uint) {
25      uint publicInt = accessibility.publicInt();
26      return publicInt;
27    }
28  }
```

Figure 8. State variable sample code

Bad Randomness

Ethereum has been exploited as a platform for a variety of blockchain applications, especially the ones that relate to random numbers, for instance the lottery and timestamps. Generating random numbers on the blockchain is technically difficult, and these numbers can simply be manipulated by attackers. Block.timestamp is one of the methods that has been adopted, but it is considered as a perilous tool, because Miners have an opportunity to select, modify, and manipulate those random numbers. Therefore, the utilize of block variables as a source should be avoided [38]. The following function can be used as the simplest random method (Figure 9).

```
30
31  // Randomness provided by this is predicatable. Use with care!
32  function randomNum() internal view returns (uint) {
33      uint random = uint(keccak256(_seed, blockhash(block.number - 1)));
34      return random;
35  }
```

Figure 9. Randomness sample code

External Contract Referencing

Reusing codes developed in Ethereum systems and interacting with other smart contracts in the network can provide plus benefits. Basically, many contracts call for a relationship with each other. Allowing such calls can help attackers use it as an attack surface. Any address in this state can be unintentionally used as a contract, the code in the address represents the type of

contract being issued. This can be dangerous, especially if the person who wrote the code is hiding something malicious in the code [32]. Avoid using external contracts for sensitive operations is an imperative safety measure, and if an external contract is required, the incoming data must be checked.

In the example below (Figure 10), the attacker was able to disable Line 77. In order to prevent this, It should be defined in the constructor section. encryptionLibrary = new Rot13Encryption();

```
71  import "Rot13Encryption.sol";
72  // encrypt your top-secret info
73  contract EncryptionContract {
74      // library for encryption
75      Rot13Encryption encryptionLibrary;
76      // constructor - initialize the library
77          encryptionLibrary = _encryptionLibrary;
78      }
79      function encryptPrivateData(string privateInfo) {
80          // potentially do some operations here
81          encryptionLibrary.rot13Encrypt(privateInfo);
82       }
83
```

Figure 10. External Conract sample code

Short Address/Parameter Attack

Parameter attack is considered as a classical SQL injection attack. As an attack method; If the EVM detects a substream when dealing with data types up to 256 bits, it adds 0 to the end of the address. The attacker is able to create this attack by removing the last zeros from an Ether address ending with 0 or multiple 0s at addresses in this situation [39].

If the necessary checks are not made, the system accepts both in the use made without the address below and the 00s at the end.

```
0xc3DC35818d54FDA1C4943bA98938cb6F46A91700
```

If the code in Figure 11 is checked for msg.data.length on line 88, it will not be accepted with ashortaddress.

```
85
86    contract NonPayloadAttackableToken {
87      modifier onlyPayloadSize(uint size) {
88       assert(msg.data.length >= size + 4);
89         _;
90        }
91    function transfer(address _to, uint256 _value) onlyPayloadSize(2 * 32) {
92     // todo
93       }
94    }
95
```

Figure 11. Short address sample code

Timestamp Dependency/Manipulation

Miners are generally considered as nodes that can interfere with transactions as an attacker. Potential danger is recognized if they have the opportunity to manipulate environmental variables and can profit from it. A miner can control the timestamp and gain an unfair advantage. They can use block numbers and the average time between blocks to estimate the current time [32], [40]. In the article published in 2019 by Mei et al. [41] Timestamp dependency was found to be the most common vulnerability [40].

In the simple roultte game below (Figure 12), miner spin can play with 1 Ether and then win for the next block. The 15 second rule should be applied. This rule is that the timestamp between two blocks within 15 seconds should not be more than 15 seconds. If the difference is phase, it should be rejected.

```
100  contract Roulette {
101      uint public pastBlockTime;
102      constructor() payable {}
103
104
105      function spin() external payable {
106          require(msg.value == 1 ether);
107          require(block.timestamp != pastBlockTime);
108          pastBlockTime = block.timestamp;
109
110          if(block.timestamp % 7 == 0) {
111            (bool sent, ) = msg.sender.call{value: address(this).balance}("");
112            require(sent, "Failed to send Ether");
113          }
114      }
115  }
```

Figure 12. Timestamp sample code

Denial Of Service (DOS)

DOS attacks are attacks that all internet-connected digital systems have to deal with from time to time. As a result of such an attack, it may be possible that the contracts become unusable for a while. This attack can freeze these contracts for an indefinite period or even indefinitely [32], [42].

In the example below, an attacker could repeatedly attack a new account, stressing the system unusable (Figure 13).

```
52 contract DistributeTokens {
53     address public owner; // gets set somewhere
54     address[] investors; // array of investors
55     uint[] investorTokens; // the amount of tokens each investor gets
56     // ... extra functionality, including transfertoken()
57     function invest() external payable {
58         investors.push(msg.sender);
59         investorTokens.push(msg.value * 5); // 5 times the wei sent
60         }
61     function distribute() public {
62         require(msg.sender == owner); // only owner
63         for(uint i = 0; i < investors.length; i++) {
64             // here transferToken(to,amount) transfers "amount" of
65             // tokens to the address "to"
66             transferToken(investors[i],investorTokens[i]);
67         }
68     }
69 }
70
```

Figure 13. DOS sample code

In order to avoid this attack, gas must be used in every account creation. It will also help in checking require(msg.sender == owner || now > unlockTime) for account.

tx.origin Authentication

Solidity has a distinctive method to check who is calling by using a function msg.sender [43]. We may use tx.origin to test who is calling as an alternative to msg.sender. As a result, an attack can be formed. A transactional attack is a form of phishing attack that can simply drain a contract of all funds [44].

```
38  contract PhishabletxOrigin {
39      address public owner;
40      constructor (address _owner) {
41          owner = _owner;
42      }
43      function () external payable {}
44      function withdrawAll(address _recipient) public {
45          require(tx.origin == owner);
46          _recipient.transfer(this.balance);
47      }
48  }
```

Figure 14. tx.origin sample code

If the control is not performed on line 45 of the code shown in Figure 14, the attacker can show himself in the state of ownership. For this reason, it should be edited as require (tx.origin == msg.sender).

6. Discussion

Several testing tools have recently emerged. Applications must be automatically checked for common security vulnerabilities from static analysis tool platforms, especially before a deployment of applications. [45], Chainsecurity [46], and Smartcheck [47] are some of them. The first written smart contract must be loaded on these systems, and a detailed result report must be examined by the system. A single application cannot detect all vulnerabilities (Table 1). Therefore, it will be efficacious to exploit different testing tools. It has been shown that while reentry attacks can be caught by all scanning programs, different results are obtained in different applications of other vulnerable parts.

When starting a new project, the first of the latest version should be preferred. Never use tx.origin for authorization checks [44]. In cases where randomness is needed, using an external source of randomness is a mandatory. Do not use the status check as the block timestamp may cause weaknesses. In addition to a frequent examination, we need to avoid a looping over especially when there is an unknown size of data structure. Using a safe library for arithmetic operations is another imperative safety measure that we must follow [38]. We should cautiously pay attention to the use of different contracts, and also avoid the codes that are obtained from untrusted sources [30].

Oyente was developed by researchers from the National University of Singapore in January 2016. Oyente can be defined as a symbolic execution tool that works directly with the Ethereum virtual machine (EVM) bytecode. Oyente is able to detect many vulnerabilities of Ethereum, especially the TheDAO bug. Currently, Oyente is available as a Docker image for easy testing and installation. It is available at https://github.com/melonproject/oyente and can be downloaded and tested [48].

Table 1. Tools matrix

Tool	Analysis Method	Source	Reentry	Timespend Manipulation	Tx.origin
Ovente	Heuristic	source code(.sol)	yes	yes	yes
Chainsecurity	Formal	Byte code and source code(.sol)	yes	no	no
Smartcheck	Analytic and Heuristic	source code(.sol)	yes	yes	yes

Unlike other examples, SmartCheck is a tool developed as a static analysis tool. SmartCheck translates Solidity source code into an XML-based intermediate representation and checks it against XPath patterns. Smartcheck is designed based on current knowledge of Solidity vulnerabilities. It has been noted that SmartCheck has limitations, as some error detection requires more sophisticated techniques such as defect analysis or even manual inspection [49].

7. Conclusion

We have done extensive research on vulnerability articles and online websites. This study identified several security vulnerabilities in the new blockchain-based smart contracts that are frequently used by many sectors. In this study, application developers have been advised to minimize their weaknesses. To improve smart contract security and its performance, we strongly suggest all users to employ safe practices and procedures. Nonetheless, in order to ensure the highest level of security, additional testings and repeating security audits on the blockchain-based smart contracts are required. Further research could be implemented if there is any present of unascertained vulnerabilities and practicable prevention.

Ethics in Publishing

There are no ethical issues regarding the publication of this study.

Conflict of Interest

The authors have no conflicts of interest to declare.

References

[1] https://bitcoin.org/bitcoin.pdf, [Accessed]: 01.07.2022.
[2] https://bitcoin.org/en/ [Accessed]: 13.10.2022.
[3] Lauslahti, K., Mattila, J. , Seppala, T. (2017) Smart Contracts How Will Blockchain Technology Affect Contractual Practices?. ETLA Reports, 1-27.
[4] Dorri, A. , Kanhere, S. S., Jurdak, R., Gauravaram, P. (2017) Blockchain for IoT security and privacy: The case study of a smart home. IEEE. International Conference on Pervasive Computing and Communications Workshops (PerCom) Workshops, 618-623.

[5] Leng, K., Bi, Y., Jing, L., Fu, H. C., Van Nieuwenhuyse, I. (2018) Research on agricultural supply chain system with double chain architecture based on blockchain technology. Future Generation Computer Systems, 86, 641-649.
[6] Raikwar, M., Mazumdar, S., Ruj, S., Sen Gupta, S., Chattopadhyay, A., Lam, K.Y. (2018) A Blockchain Framework for Insurance Processes. IFIP International Conference on New Technologies, Mobility and Security (NTMS), 1-4.
[7] Amofa, S., Sifah, E. B., Kwame O. Agyekum, B. Abla, S., Xia, Q., Gao, J. (2018) A Blockchain-based Architecture Framework for Secure Sharing of Personal Health Data. IEEE. International Conference on e-Health Networking, Applications and Services (Healthcom), 1-6.
[8] Ma, Z., Huang, W., Gao, H. (2018) Secure DRM Scheme Based on Blockchain with High Credibility. Chinese Journal of Electronic, 27(5), 1025-1036.
[9] Azaria, A., Ekblaw, A., Vieira, T., Lippman, A. (2016) MedRec: Using Blockchain for Medical Data Access and Permission Management. International Conference on Open and Big Data, 25-30.
[10] Chen, B., Tan, Z., Fang, W. (2018) Blockchain-Based Implementation for Financial Product Management. International Telecommunication Networks and Applications Conference,1-3.
[11] http://arxiv.org/abs/2007.00286 [Accessed]: 01.07.2022.
[12] Kemmoe, V. Y., Stone, W., Kim, J., Kim, D., Son, J. (2020) Recent Advances in Smart Contracts: A Technical Overview and State of the Art. IEEE Access, 8, 117782-117801.
[13] 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.
[14] https://nakamotoinstitute.org/static/docs/micropayments-and-mental-transaction-costs.pdf [Accessed]: 01.07.2022.
[15] https://www.inve topedia.com/terms/s/smart-contracts.asp [Accessed]: 12.07.2022.
[16] https://ordina-jworks.github.io/blockchain/2017/05/10/Blockchain-Introduction.html#smart-contracts [Accessed]: 13.09.2022.
[17] Ribeiro, S. L.I., Barbosa, A. P. (2020) Risk Analysis Methodology to Blockchain-based Solutions. Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS), 59-60.
[18] http://arxiv.org/abs/1904.03487 [Accessed]: 13.11.2022.
[19] Xu, J. J. (2016) Are blockchains immune to all malicious attacks?. Financ Innovation, 2(1), 25.
[20] Wang, S., Wang, C., Hu, Q. (2019) Corking by Forking: Vulnerability Analysis of Blockchain. IEEE Conference on Computer Communications, 829-837.
[21] Dey, S. (2018) Securing Majority-Attack in Blockchain Using Machine Learning and Algorithmic Game Theory: A Proof of Work. Computer Science and Electronic Engineering, 7-10.
[22] Ramezan, G., Leung, C., Wang, J. Z. (2018) A Strong Adaptive, Strategic Double-Spending Attack on Blockchains. International Conference on Internet of Things and IEEE Green Computing and Communications and IEEE Cyber, Physical and Social Computing and IEEE Smart Data, 1219-1227.
[23] Zhao, X., Chen, Z., Chen, X., Wang, Y., Tang, C. (2017) The DAO attack paradoxes in propositional logic. International Conference on Systems and Informatics, 1743-1746.
[24] https://www.wired.com/2016/06/50-million-hack-just-showed-dao-human/ [Accessed]: 23.10.2022.
[25] https://www.investopedia.com/news/ethereum-smart-contracts-vulnerable-hacks-4-million-ether-risk/ [Accessed]: 13.10.2022..

[26] https://www.findlaw.com/legalblogs/technologist/reports-ethereum-smart-contracts-are-far-from-secure/ [Accessed]: 14.10.2022.
[27] https://www.stateofthedapps.com/stats [Accessed]: 14.10.2022.
[28] https://arxiv.org/ [Accessed]: 13.10.2022..
[29] https://ieeexplore.ieee.org/Xplore/home.jsp [Accessed]: 13.10.2022.
[30] https://blog.sigmaprime.io/solidity-security.html#reentrancy [Accessed]: 14.10.2022.
[31] Atzei, N., Bartoletti, M., Cimoli, T. (2017) A Survey of Attacks on Ethereum Smart Contracts (SoK). Principles of Security and Trust, 164-186.
[32] Dika, A., Nowostawski, M. (2018) Security Vulnerabilities in Ethereum Smart Contracts. IEEE International Conference on Internet of Things and IEEE Green Computing and Communications and IEEE Cyber, Physical and Social Computing and IEEE Smart Data, 955-962.
[33] Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A. (2016) Making Smart Contracts Smarter. ACM SIGSAC Conference on Computer and Communications Security, 254-269.
[34] https://consensys.github.io/smart-contract-best-practices/attacks/ [Accessed]: 13.10.2022.
[35] http://arxiv.org/abs/1902.06710 [Accessed]: 15.07.2022.
[36] Grech, N., Kong, M., Jurisevic, A., Brent, L., Scholz, B., Smaragdakis, Y. (2018) MadMax: surviving out-of-gas conditions in Ethereum smart contracts. ACM Programing Language, 1-27.
[37] Nan, Y., Yang, Z., Wang, X., Zhang, Y., Zhu, D., Yang, M. (2018) Finding Clues for Your Secrets: Semantics-Driven, Learning-Based Privacy Discovery in Mobile Apps. Network and Distributed System Security Symposium.
[38] https://yos.io/2018/10/20/smart-contract-vulnerabilities-and-how-to-mitigate-them/#vulnerability-all-data-is-public/ [Accessed]: 14.10.2022.
[39] https://medium.com/huzzle/ico-smart-contract-vulnerability-short-address-attack-31ac9177eb6b [Accessed]: 13.10.2022.
[40] Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., Alexandrov, Y. (2018) SmartCheck: Static Analysis of Ethereum Smart Contracts. EEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain, 9-16.
[41] Mei, X., Ashraf, I., Jiang, B., Chan, W. K. (2019) A Fuzz Testing Service for Assuring Smart Contracts. IEEE International Conference on Software Quality, Reliability and Security Companion, 544-545.
[42] Saad, M., Njilla, L., Kamhoua, C., Kim, J., Nyang, D., Mohaisen, A. (2019) Mempool optimization for Defending Against DDoS Attacks in PoW-based Blockchain Systems. IEEE International Conference on Blockchain and Cryptocurrency, 285-292.
[43] https://consensys.github.io/smart-contract-best-practices/development-recommendations/solidity-specific/tx-origin/ [Accessed]: 14.10.2022.
[44] https://medium.com/coinmonks/solidity-tx-origin-attacks-58211ad95514[Accessed]: 14.10.2022.
[45] https://github.com/enzymefinance/oyente [Accessed]: 14.10.2022.
[46] https://chainsecurity.com/audits/ [Accessed]: 14.10.2022.
[47] https://smartdec.net/ [Accessed]: 14.10.2022.
[48] https://medium.com/haloblock/how-to-use-oyente-a-smart-contract-security-analyzer-solidity-tutorial-86671be93c4b [Accessed]: 14.10.2022.
[49] Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., Alexandrov, Y. (2018) SmartCheck: Static Analysis of Ethereum Smart Contracts. IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain, 9-16.