DOI: https://doi.org/10.15276/hait.06.2023.4
UDC 004.4'24

Requirements for the development of smart contracts and an overview of smart contract vulnerabilities at the Solidity code level on the Ethereum platform

Nataliia O. Komleva[1)]
ORCID: http://orcid.org/0000-0001-9627-8530, komleva@op.edu.ua. Scopus Author ID: 57191858904
Oleksandr I. Tereshchenko[1)]
ORCID: http://orcid.org/0000-0003-4510-5255, alexandr.tereschenko2014@gmail.com. Scopus Author ID: 57705566400
[1)] Odessa National Polytechnic University, 1, Shevchenko Ave. Odessa, 65044, Ukraine

ABSTRACT

The article is devoted to the consideration of automated decentralized programs on the blockchain, which are a modern tool for processing transactions without the help of a trusted third party. The purpose of the study is to generalize and systematize information on the requirements for smart contracts, as well as review the vulnerabilities of smart contracts at the Solidity code level. The blockchain architecture was studied and the advantages of smart contracts compared to conventional contracts were determined, namely: risk reduction, reduction of administration and maintenance costs, and improvement of business process efficiency. A thorough analysis of current literature has been carried out and the current problems faced by users and developers of smart contracts have been identified. It is noted that the process of developing smart contracts is not sufficiently standardized and it is advisable to create a system of recommended requirements for smart contracts used in various subject areas. The requirements for smart contracts have been collected and analyzed for areas related to healthcare, education, business, project management, data analysis, software development, trading, logistics, and jurisprudence. It is determined that the mandatory requirements for all these subject areas are security, process transparency, determination of conditions and criteria for success, and automation of work. The rest of the requirements are analyzed and the concepts of the measure of coincidence and uniqueness of requirements for a particular subject area based on the corresponding functions are introduced. The coincidence and uniqueness measures were calculated for the considered subject areas. The proposed measures will allow in the future to obtain a quantitative assessment of templates for gathering requirements for programs, taking into account the used subject area. The article reviews and systematizes the types of vulnerabilities of smart contracts at the level of Solidity code on the Ethereum platform. The best practices to avoid such vulnerabilities and possible examples of their exploitation by attackers are identified. It has been shown that increasing the reliability of smart contracts will help increase trust in the blockchain among users.

Keywords: Blockchain; smart contract; requirements; coincidence measure; uniqueness measure; Ethereum; vulnerability; transaction

***For citation*: Komleva N. O., Tereshchenko O. I. "Requirements for the development of smart contracts and anoverview of smart contract vulnerabilities at the Solidity code level on the Ethereum platform". *Herald of Advanced Information Technology.* 2023; Vol.6 No.1: 54–**68 . DOI: https://doi.org/10.15276/hait.06.2023.4**

1. INTRODUCTION

Smart contracts are automated decentralized applications on the blockchain that describe the terms of the agreement between buyers and sellers, reducing the need for intermediaries and arbitration.

Smart contracts were originally proposed to digitize and automate legal contracts, but later in the context of blockchain, they came to mean scripts of code executed by nodes in the blockchain network [1, 2]. Certain misunderstandings related to the divergence of concepts have significantly slowed down the creation of smart contract standards [3].

The use of smart contracts simplifies work in many areas of business, increasing trust between business partners and significantly reducing costs. However, frequent incidents related to the security of smart contracts not only lead to huge economic losses, but also destroy trust in the blockchain.

According to statistics from Defiyield [4], the economic losses caused by security issues in smart contracts exceeded $47 billion in 2022, which is 500% more than in 2021. For example, in June 2016, hackers exploited the reentrancy vulnerability of "The DAO" contract to steal about $60 million worth of Ether (the digital currency of Ethereum). In July 2017, due to a vulnerability in the *delegatecall* function of the Parity Multisig Wallet contract, Ether worth almost $300 million was frozen. In April 2018, the attackers exploited an integer overflow vulnerability in the BeautyChain contract to issue an unlimited number of BEC tokens, which led to a

drop in the price of BEC tokens to zero. In May 2019, hackers hacked Binance Exchange, which led to the theft of more than 7000 BTC.

To create secure smart contracts, developers should be aware of the most common smart contract vulnerabilities. This article provides an overview of the most common smart contract vulnerabilities, as well as examples of how these vulnerabilities are exploited by attackers and approaches to avoid these vulnerabilities in the Solidity code of smart contracts on the Ethereum platform.

Innovative blockchain technology can be used to develop new types of services and platforms in various subject areas. For each subject area, the requirements for blockchain-based developments may be different and aimed at achieving specific goals and objectives.

Thus, studying the requirements for smart contracts for different subject areas and a detailed review of possible vulnerabilities at the Solidity code level is an urgent task.

The purpose of the study is to summarize and systematize information on the requirements for smart contracts, as well as to review the vulnerabilities of smart contracts at the Solidity code level.

To achieve the purpose, the following tasks should be solved:

– to study the blockchain architecture and the differences between the blockchain transaction model and the traditional transaction model;

– to analyze the requirements for smart contracts and determine the degree of specificity of the requirements through the measure of their coincidence and uniqueness depending on the scope of the smart contract;

– to systematize the types of vulnerabilities of smart contracts at the level of the Solidity code;

– to perform an experimental study of the considered vulnerabilities.

2. LITERATURE REVIEW AND STATEMENT OF THE PROBLEM

A blockchain is a system of distributed software that allows transactions to be processed without the help of a trusted third party. A blockchain is a sequence of records that are grouped into blocks, hashed, and linked to the previous block. Fig. 1 displays the structure of a blockchain.

A transaction is a data storage operation in a blockchain, during which crypto assets or other information are transferred between participants. Due to the characteristics of a blockchain, such as transparency, decentralization and protection from tampering, trust is ensured by users, since it is almost impossible to forge the transactions stored in a blockchain, as all past transactions are verifiable and traceable.

Smart contracts were proposed in the 1990s by Nick Szabo [5] and became one of the most successful applications of a blockchain. Contract clauses in a smart contract are executed automatically when predetermined conditions are met. Smart contracts consisting of transactions are stored, distributed, and updated on distributed blockchains. In contrast, conventional contracts must be executed by a verified third party centrally, which, as a result, leads to long execution times and additional costs.

The following advantages of smart contracts over conventional contracts can be identified:

1. Risk reduction. Due to the immutability of the blockchain, smart contracts cannot be arbitrarily changed after their creation. In addition, all transactions stored and duplicated across the entire distributed blockchain system can be tracked and verified.

2. Reduction of administration and maintenance costs. Blockchain ensures trust in the entire system using distributed consensus mechanisms without the use of an intermediary.

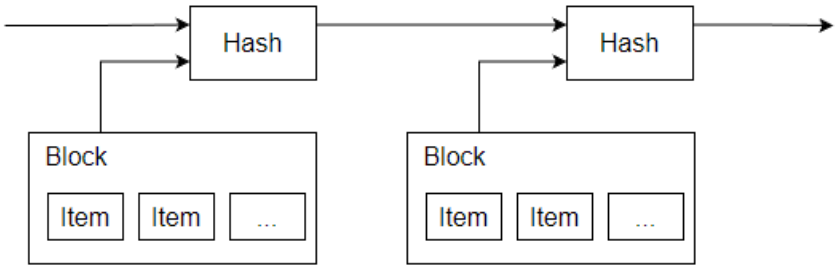

Fig. 1. **Structure of a blockchain**
Source: **compiled by the authors**

3. An increase of the efficiency of business processes. Elimination of dependence on intermediaries can significantly increase the efficiency of business processes.

Ethereum is the first blockchain platform that supports smart contracts. It uses the Solidity programming language to develop smart contracts and provides a large number of application programming interfaces (API). Ethereum accounts are divided into two categories: externally owned accounts (EOA), which have a balance and can make smart contract calls, and internal accounts, which, unlike external accounts, have a balance and a smart contract code that can be executed in addition to calls to other smart contracts. Smart contracts are executed by the Ethereum Virtual Machine (EVM), which is a Turing-complete, stackable virtual machine for executing smart contract bytecodes.

The authors [5] note that the vast majority of smart contracts lack any formal specification necessary to establish their correctness. The code does not always do what it was originally intended to do, which leads to significant financial losses, so static verification of smart contracts is necessary [6].

Standards have appeared in the literature only recently and contain the properties of smart contracts, basic requirements for their development and some architectural solutions [7]. According to the PMBOK project management guide, stakeholders influence projects, performance, and results [8]. Thus, the involvement of stakeholders and relevant focus groups helps to identify the most important project requirements and improves the delivery of their value to the end user. Prioritization of requirements is an extremely important work aimed at achieving maximum value. According to the BABOK professional standard for business analysts, priority can reflect the relative value of a requirement or the order in which it will be implemented [9]. At the same time, prioritization is an ongoing process in which priorities can change with the changing context.

Smart contracts have proven themselves and are used in many fields – medical, educational, legal sphere, project management, etc. It is advisable to use them in solving other complex intellectual tasks [10, 11].

A literature review of the modern use of smart contracts has shown that there are a number of issues that need to be addressed.

The article [12] discusses the problems of preserving confidentiality in the field of medical insurance. It is noted that in medical insurance contracts it is necessary to enter information about the patient's condition as the logic of the decision in order to initiate further implementation. At the same time, since the blockchain is a closed network, it lacks a secure network environment for data interaction with the outside world. Information about privacy can still be obtained by analyzing the results of the transaction, since contract states are publicly available. At the same time, the authors [13] propose a blockchain security system that protects medical data collected from the Internet of Medical Things (IoMT) system with a modified SHA-256 hashing algorithm, using the code length algorithm to compress data.

The work [14] studies the algorithms and features of the implementation of smart contracts, which solve the issues of accountability, transparency, traceability and audit due to the possible overproduction and insufficient consumption of medication stocks in the existing supply chain systems in the field of healthcare.

Specialists who are engaged in the application of smart contracts for the learning process face many problems related to trust in the course, certification of credits and certificates, privacy of students and sharing of courses. The paper [15] contains research on the development of smart contracts that can store educational records in a reliable and distributed manner, provide reliable digital certificates, implement the exchange of educational resources, and protect intellectual property using data encryption.

Researchers in the paper [16] raise the issue of fraud in the issuance of academic certificates and diplomas, as well as the verification of resumes, which have long been a problem in the academic community. A model is proposed, which includes a scheme with several signatures with the regulation of the issuance of academic documents in a decentralized manner. At the same time, anyone in the world can verify the authenticity of the document by activating the corresponding function of the smart contract, thereby eliminating any possibility of fraud.

Legal contracts can potentially contain ambiguities and lead to misinterpretations. Smart contracts used for this purpose are designed to identify such ambiguities and quantify them [17]. Unifying models which encapsulate the main components of legal smart contracts are proposed to develop and verify the characteristics of smart contracts [18].

In the paper [19], the authors study the problems of managing scientific projects and note

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

the tendency to increase the number of scientific research projects. Due to the lack of a standardized and unified research project management programs, many projects are delayed or even failed, and project fund management is confused. In addition, the output results are limited and the actual conversion rate is low. Management of scientific research projects according to blockchain consortium, smart contract and IPFS system allows to cope with two main problems of traditional scientific project management: breach of contract and confidentiality. The paper [20] examines the advantages and disadvantages of using blockchain in project management in the commercial sphere and proposes a management model to replace manual operations.

A number of studies have focused on the use of blockchain technology and smart contracts for managing agile projects using Scrum or Lean-Kanban processes. This allows to delegate the product owner's responsibilities for confirming the correctness of the results to one or more smart contracts deployed on the Ethereum blockchain and written in Solidity [21]. At the same time, the customer agreement can also allow smart contracts to automatically enable payments and impose penalties based on the result. In this way, the product owner may be relieved of his duties and responsibilities, allowing resources to be allocated to more profitable and productive tasks.

From a technical point of view, the vulnerabilities of Ethereum smart contracts can be divided into three levels, that is the Solidity code level, the EVM execution level, and the block dependency level. The NCC Group has created an analog of the OWASP TOP 10 project for smart contract vulnerabilities, called the Decentralized Application Security Project (DASP) TOP 10 [22]. The project presents the 10 most critical vulnerabilities of smart contracts. In addition, there is a register of smart contract vulnerabilities, the SWC Registry, which is constantly updated [23]. So far, more than 30 discovered vulnerabilities have been added to it.

In the papers [24, 25], the authors introduced an error detection tool that detects various types of Solidity code level vulnerabilities in smart contracts. Mutated contracts were used to evaluate the effectiveness of various analysis tools. In the paper [26], experiments are described using 1838 real contracts, from which 12866 modified contracts were created by artificially seeding 8 different types of vulnerabilities. The technique's effectiveness in detecting vulnerabilities was evaluated and compared with five existing popular analysis tools – Oyente, Securify, Maian, SmartCheck and Mythril.

The following conclusions can be drawn from the above analysis of the literature:

– the development and use of smart contracts is a very promising area and provides objective advantages when applied in various subject areas;

– the process of developing smart contracts is not sufficiently standardized; there is no system of recommended requirements for creating effective smart contracts for different subject areas, so the creation of such a system would be advisable;

– it is necessary to take into account various types of vulnerabilities that can interfere or even make it impossible to use smart contracts during the development of smart contracts;

– research and systematization of smart contract vulnerabilities at the Solidity code level is an urgent task.

3. TYPES OF SMART CONTRACTS

There are several types of smart contracts depending on their application [27].

1. *Legal smart contracts*. These contracts are legally binding and the parties are obliged to fulfill their contractual obligations. If they fail to fulfill the obligations, they may face legal consequences. Such smart contracts are used by cryptocurrency exchanges, DeFi and Game-Fi projects, and various blockchain platforms, from NFT markets to metauniverses and real estate trading.

2. *Decentralized Autonomous Organizations* (DAO). Decentralized autonomous organizations are an organizational form in which the coordination of participants' activities and resource management takes place in accordance with a pre-agreed and formalized set of rules, which are monitored automatically. The rules of DAO operation are described in smart contracts. Records of DAO financial transactions and the programmatic rules of such contracts are stored on the blockchain. Examples of DAOs include the Decentraland, Uniswap, Polkadot, and MakerDAO governance protocols. According to the rules of these projects, they are managed by token holders who can make various proposals (for example, determine the structure of commissions) and vote for them. In this case, DAO smart contracts are responsible for voting and vote counting.

3. *Smart contracts of application logic.* These contracts include program code that is usually synchronized with other smart contracts. They also provide a link between Internet of Things (IoT) devices and blockchain technology. In addition, such

smart contracts can handle communication between the blockchain and blockchain oracles.

4. LIFE CYCLE AND REQUIREMENTS FOR SMART CONTRACTS

Smart contracts perform different tasks at different stages of the life cycle, and each stage can lead to certain security issues due to technical inexperience of developers. Thus, the improvement of security of smart contracts requires developers and users to have a deep understanding of the characteristics of each phase of the smart contract lifecycle.

The entire life cycle of smart contracts consists of four consecutive phases, namely creation, deployment, execution and destruction of a smart contract.

1. *Creation of a smart contract.* Several involved parties first negotiate the obligations, rights and prohibitions in the contract. After careful discussions and negotiations, an agreement can be reached. Lawyers or consultants will help the parties draft the initial contractual agreement. Programmers then convert this agreement written in natural languages into a smart contract written in computer languages, including declarative languages. Similar to computer software development, the procedure of creating a smart contract consists of design, implementation and testing. It is worth noting that the creation of smart contracts is an iterative process involving several rounds of negotiations and iterations with stakeholders, namely lawyers and software developers. Currently, there are more than 40 platforms that support the deployment of smart contracts, all of which have corresponding contract development languages. In addition to Solidity, there are more than 10 programming languages for developing smart contracts, such as Vyper and Idris. The developed smart contract code must be compiled. The Ethereum virtual machine converts the byte code into the corresponding operation code to execute the smart contract. Once compiled by the compiler, an Ethereum smart contract will also create an application binary interface (ABI) for the blockchain that enables other smart contracts and users to interact with that smart contract.

2. *Deployment of a smart contract*. Smart contracts run on the blockchain platform and need to be deployed on the blockchain to synchronize them with each node for invocation. The contract is deployed by sending a transaction to the blockchain with an empty recipient containing a byte code and other information, and is packaged by the miner into a block and returns the address of the created smart contract. This address is a unique identifier of the contract and is used to invoke or access the contract.

3. *Execution of a smart contract*. The invocation of a smart contract requires the use of the contract address and the application binary interface (ABI). There are two ways to invoke a smart contract: by sending a “transaction” or a “message”. A “transaction” is defined in the Ethereum white paper as a string of data signed by an external account (EOA) [28], so an external account can invoke a contract by sending a transaction call. The definition of a “message” in the Ethereum Yellow Book is as follows: it is data and Ether sent between two accounts [28]. A smart contract calls other contracts through “messages”. The “messages” are transmitted internally and are not synchronized with the blockchain, so calls between contracts are not recorded in blockchain.

4. *Destruction of a smart contract*. Smart contracts implement a self-destruct function that needs to be written into the contract during development. Since blockchain data cannot be deleted, a destroyed contract still exists in the blockchain, but it cannot be invoked again and its status is marked as destroyed.

Fig. 2 displays the life cycle of a smart contract.

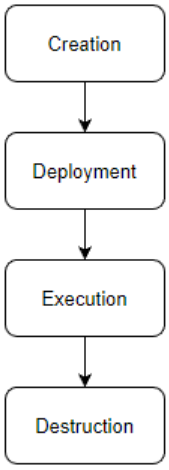

Fig. 2. **Life cycle of a smart contract**
Source: **compiled by the authors**

The requirements for creating smart contracts may depend on the specific application and the desired functionality. According to the existing standard, the inherent properties of smart contracts are immutability, transparency, and automatic execution. The functional interaction of a smart contract with other smart contracts, scalability, and synchronization for independent and connected smart contracts are also important [7].

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

However, depending on the specific task, the creation of a smart contract may require additional requirements. Therefore, before creating a smart contract, it is necessary to conduct a detailed analysis of requirements and functionality to determine the optimal contract configuration.

As part of this study, focus groups were created to determine the specifics of the requirements for smart contracts depending on the subject area of their application *S*:

$$S = \{S_1, S_2, \ldots, S_9\}, \quad (1)$$

where S_1 is medicine; S_2 is education; S_3 is business; S_4 is project management; S_5 is data analysis; S_6 is software development; S_7 is trading; S_8 is logistics; S_9 is legal sphere.

The total set of potential critical requirements for smart contracts empirically identified by focus groups for subject areas *S*:

$$R = \{R_1, \ldots, R_{34}\}, \quad (2)$$

where R_1 is security; R_2 is transparency of processes; R_3 is determination of the conditions and criteria for the success of the smart contract; R_4 is automation of work; R_5 is privacy; R_6 is compliance with legislation; R_7 is ability to track changes in medical data and their sources; R_8 is flexibility to support diverse needs; R_9 is interoperability to ensure integration into the subject area; R_{10} is risk management to prevent unauthorized actions; R_{11} is reduction of costs associated with the fulfillment of contractual obligations; R_{12} is integration with business processes and systems; R_{13} is standardization to simplify and accelerate the creation process and ensure compatibility with other systems; R_{14} is distribution of tasks within a specific project; R_{15} is monitoring of project implementation; R_{16} is data analytics for process optimization; R_{17} is – data quality assurance, including data verification and validation, as well as protection against possible manipulation; R_{18} is data versatility in terms of supporting work with different types of data for the ability to analyze various information flows; R_{19} is scalability in processing large amounts of data and the ability to work with distributed systems; R_{20} is data access control to prevent unauthorized access; R_{21} is support of various data analysis algorithms to select the optimal method for a particular task; R_{22} is transportability as the ability to transfer to other blockchains or distributed systems; R_{23} is data processing efficiency with minimization of time and resources; R_{24} is reporting on results with the ability to track them and make further decisions; R_{25} is reliability and stability to guarantee contract fulfillment and data security; R_{26} is consideration of intellectual property rights and the possibility of its protection; R_{27} is authorization and authentication of users to prevent unauthorized access to data; R_{28} is ease of use and intuitive interface to accelerate adaptation to use; R_{29} is the ability to control the use of data and receive income from their monetization; R_{30} is test acceleration to automatically check code for bugs and issues; R_{31} is acceleration of the speed of operations to reduce risks and increase the efficiency of operations; R_{32} is support of different types of assets such as currencies, securities, goods and services; R_{33} is tracking as an opportunity to track the location and status of goods/services at each stage of the logistics chain, which allows to control the delivery process and respond in a timely manner to possible delays and problems; R_{34} is audit control to establish compliance with legal requirements.

Thus, the total number of requirements is very large, and collecting requirements for the development of blockchain-based platforms and services can be a difficult task. Creating templates for collecting requirements that would include all the requirements of the set $R(S_i)$ for a particular subject area S_i, $i = 1..|S|$ can help determine the basic needs and goals of users. The set $R(S_i)$ must be finite, necessary, sufficient, and comprehensive.

Let's define the general requirements for all considered subject areas:

$$R(S_1) \cap R(S_2) \ldots \cap R(S_9) = \{R_1, R_2, R_3, R_4\}, \quad (3)$$

that is, the general requirements for smart contracts of all areas are security, transparency, definition of conditions and success criteria, and automation of work.

Now let's analyze the rest of the requirements. Let's introduce the function of matching requirements m_{ab}^r for subject areas $a \in S, b \in S$, $a \neq b$, which is calculated as:

$$m_{ab}^r = \begin{cases} 1, \text{if r is present in both a and b} \\ 0 \text{ otherwise} \end{cases} \quad (4)$$

Let's introduce the concept of the measure of coincidence of requirements M_a^r for a specific subject area *a*:

$$M_a^r = \sum_{r=1}^{|R|} \sum_{b=1}^{|S|} m_{ab}^r, a \neq b, \quad (5)$$

where $|R|$ is the power of the set of requirements, $|S|$ is the power of a set of subject areas.

The greater the measure of uniqueness M_a^r with other subject areas, the more their templates for requirements gathering should overlap.

Let's introduce the requirement uniqueness function u_a^r for the subject area $a \in S$, which is calculated as:

$$u_a^r = \begin{cases} 1, \text{if the r is present only in a} \\ 0 \text{ otherwise} \end{cases} \quad (6)$$

Let's introduce the concept of the measure of uniqueness of requirements U_a^r for a certain subject area $a \in S$:

$$U_a^r = \sum_{r=1}^{|R|} u_a^r, \quad (7)$$

where $|R|$ is the power of the set of requirements.

The measure of the uniqueness of requirements U_a^r will reflect the number of requirements present only in the template of the corresponding subject area. It should be noted that the mechanism and algorithms for generating requirements templates are not the subject of this article.

Table 1 summarizes the results of the analysis of requirements for various subject areas. The second column contains the elements of the requirement set $R(S_i) \in R$ for the respective subject area. The last two columns contain the values of the coincidence and uniqueness measures calculated according to (5) and (7).

Table 1. **Results of the requirements analysis**

Subject area	The most important requirements	The measure of coincidence of requirements	The measure of uniqueness of requirements
S_1	R_1-R_8	45	0
S_2	R_1-R_5, R_7-R_{10}	47	0
S_3	R_1-R_6, R_{10}-R_{13}	49	0
S_4	R_1-R_4, R_8, R_{10}-R_{16}	48	2
S_5	R_1-R_5, R_8, R_9, R_{12}, R_{17}-R_{29}	50	11
S_6	R_1-R_4, R_{13}, R_{30}	36	1
S_7	R_1-R_4, R_7, R_{12},R_{13}, R_{15},R_{19},R_{25}, R_{31}, R_{32}	46	2
S_8	R_1-R_4, R_8, R_{12},R_{13}, R_{19}, R_{33}	46	1
S_9	R_1-R_7, R_9, R_{25},R_{34}	45	1

Source: **compiled by the authors**

It is evident that there are significant differences in the definition of requirements for smart contracts depending on the chosen subject area. This suggests the need to define mechanisms and templates for collecting requirements based on the subject area used.

5. REVIEW AND DISCUSSION OF THE TYPES OF VULNERABILITIES OF SMART CONTRACTS AT THE LEVEL OF SOLIDITY CODE

1. *Reentrancy vulnerability*. Reentrancy vulnerability is the most common vulnerability of smart contracts [29]. The execution of smart contracts is not atomic and consistent, which leads to certain security gaps. Attackers can re-enter the called function during the current program execution [30]. Like most programming languages, smart contracts use cross-functional or cross-contract calls to process business logic. But the difference is that such calls aim to transfer some valuable assets. Calling the *transfer* function in the sender's contract will inevitably trigger the fallback function in the recipient's contract. When a smart contract performs a cross-contract money transfer operation, attackers can intercept such an external call and perform some malicious operations. An example of such an operation is when an attacker injects malicious code into his fallback function, which implements a recursive entry into the victim's contract to re-call the *transfer* function to steal ether. The reentrancy vulnerability led to the largest security incident in the history of smart contracts (the attack on “The DAO”), which not only resulted in a loss of almost $60 million but also caused the Ethereum hardfork. Fig. 3 displays the scheme of a reentrancy attack on “The DAO”.

One approach to prevent the reentrancy vulnerability is to change the user's balance before performing any interactions with other smart contracts.

The following code snippet demonstrates examples of vulnerable and secure functions:

```
// vulnerable contract
function vulnerableWithdraw() public {
    uint256 amount =
balances[msg.sender];

require(msg.sender.call.value(amount)(
));
    balances[msg.sender] = 0;
}
```


ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

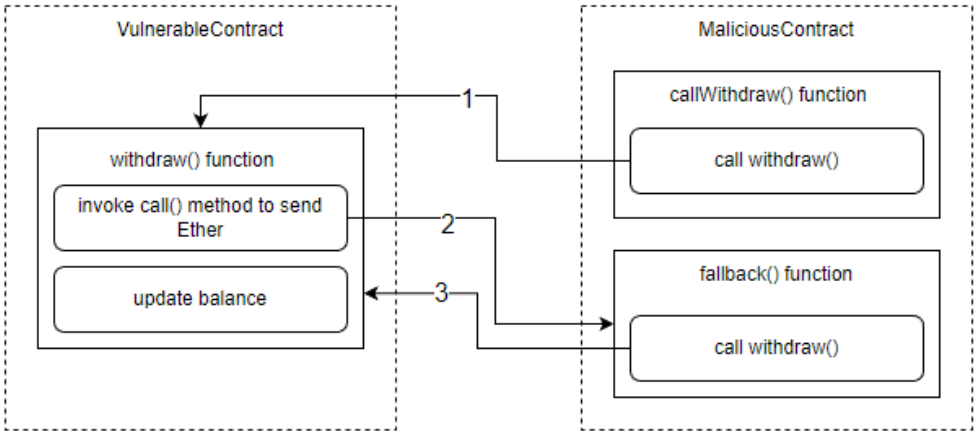

Fig. 3. **Reentrancy attack**
Source: **compiled by the authors**

```
// secure contract
function withdraw() external {
    uint256 amount =
balances[msg.sender];
    balances[msg.sender] = 0; //
change the balance before calling the
call method

require(msg.sender.call.value(amount)(
));
}
```

Since the user's balance in the secure function is set to zero in advance, when the *withdraw* method is recursively called, the cryptoassets will not be re-sent to the attacker's address.

2. *Integer overflow/underflow vulnerability*. Integer overflow/underflow is a common vulnerability in programs. There are three types of integer overflow/underflow vulnerabilities in smart contracts, namely multiplication overflow, addition overflow, and subtraction underflow. In the source code of smart contracts, integers are treated as unsigned integers with a fixed size. Obviously, if an integer variable exceeds a certain range, an integer overflow error will occur.

Ethereum smart contracts are written in high-level languages such as Solidity, which support the range of integers from *uint8* to *uint256*. For example, if a number is of type *uint8*, its value is stored as an 8-bit unsigned number in the range from 0 to 256−1. If a value outside of this range is assigned to a variable of type *uint8*, the Ethereum Virtual Machine (EVM) will automatically truncate the leading digits. For example, let's imagine that we have a variable of type *uint8* that stores the value 255 in the decimal system or 11111111 in the binary system. If we want to add 1 to this variable, then in the binary system we get the number 100000000, which takes 9 bits. Since our data type takes only 8 bits, the leading bit will be discarded. Thus, we will get the value of the variable 0, not 256 as planned.

Unlike other applications, the losses caused by the integer overflow vulnerability in smart contracts are huge and irreparable. For example, the integer overflow vulnerability was exploited by attackers who infinitely copied BEC tokens, causing the BEC token price to drop to zero. The *mintToken* function of the Coinstar (CSTR) token smart contract has an integer overflow that allows the owner of the contract to set any value of the balance of an arbitrary user. Currently, to prevent integer overflow in smart contracts, developers need to not only check code manually, but also use the SafeMath library to check arithmetic logic, or the Solidity compiler, starting with version 0.8, which automatically takes care of checking code for integer overflow/underflow vulnerabilities.

3. *Access control vulnerability*. The main reason for the access control vulnerability is that developers forget to explicitly set access restrictions to functions in the smart contract, which allows attackers to use functions or variables that they should not have access to. Access control vulnerabilities usually occur at two levels: the code level and the logical level. At the code level, there are four types of access restrictions to smart contract functions and variables, namely public, private, external, and internal. At the logical level, modifiers are used to restrict access rights to functions in smart contracts, such as *onlyOwner* and *onlyAdmin*. Functions without restriction modifiers indicate that anyone has access to them, which can lead to manipulation of key functions by attackers, compromising the security of smart contracts.

4. *Vulnerability of improperly handled exception from an external function*. Correct exception handling is one of the most important

mechanisms for improving the reliability and resilience of applications. In general, a smart contract handles abnormal behavior with a rollback, namely stopping the current execution of the contract, restoring the previous state, and returning an error ID. However, the smart contract which calls the external function may not be able to retrieve the exception information from the called contract. In most cases, an exception thrown in an external smart contract function is returned to the contract that called it. However, some low-level function calls, such as *call*, *callcode*, *send*, and *delegatecall*, only return *false* and not the exception itself. Therefore, it is not possible to judge the successful execution of a contract based on whether an exception was generated in the smart contract. As a result, the execution of the contract will continue even if an exception occurs in the external contract, which will lead to a violation of the program logic and it can be exploited by attackers. Therefore, it is necessary to process the return value when calling such functions. The following code snippet demonstrates a vulnerable smart contract that does not handle the return value of the *call* method when the external smart contract's *malicious_func* method is called:

```
pragma solidity 0.4.25;
contract VulnerableContract {
    address MaliciousContract;
    function callnotchecked() public {
        // call the vulnerable
function of the contract
VulnerableContract, which returns
false

MaliciousContract.call(bytes4(sha3("ma
licious_func()")));
    }
}
```

The given code fragment shows an example of an attacker's smart contract, which generates an exception when the *malicious_func* method is called by third-party contracts:

```
pragma solidity 0.4.25;
contract MaliciousContract {
    function malicious_func() public {
        require(false); // throw an
exception
    }
}
```

One approach to dealing with this vulnerability is to wrap low-level functions, such as *call*, with a *require* function.

5. *Denial of service vulnerability*. Denial of service (DoS) is a common vulnerability of Ethereum smart contracts [31]. Attackers typically exploit this vulnerability to break the original program logic and make the contract unable to provide the usual services for some time or even permanently. There are three types of DoS attacks against smart contracts.

1) A DoS attack launched by an unexpected call of the *revert* function in an external function. The smart contract will be susceptible to a DoS attack if it tries to send funds to the user, and the further operation of the smart contract depends on the success of this operation. The problem can arise if funds are sent to a smart contract created by attackers, as they can simply create a callback function that cancels all payments.

A code snippet vulnerable to a DoS attack launched by an unexpected call of the *revert* function in an external function:

```
contract Auction {
    address currentLeader;
    uint highestBid;

    function bid() payable {
        require(msg.value >
highestBid);
 // Payment to the previous leader is
not made
        // due to a call of the revert
function in an external contract

require(currentLeader.send(highestBid)
);

        currentLeader = msg.sender;
        highestBid = msg.value;
    }
}
```

It is easy to see in this example that an attacker makes a bid from a smart contract with a callback function that cancels all payments. Therefore, the funds will never be refunded to him and no one will ever be able to make a higher bid.

An effective way to deal with such a vulnerability is to separate the refund operation into a separate function that the participant will need to call independently. An example of secure code is provided in the following code snippet:

```
contract Auction {
    address highestBidder;
    uint highestBid;
    mapping(address => uint) refunds;

    function bid() payable external {
        require(msg.value >=
highestBid);

        if (highestBidder !=
address(0)) {
 // record in a separate array the
funds that should be refunded
```


ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

```
            refunds[highestBidder] +=
highestBid;
        }

        highestBidder = msg.sender;
        highestBid = msg.value;
    }

    // the user calls the method to
get the refund
    function withdrawRefund() external
{
        uint refund =
refunds[msg.sender];
        refunds[msg.sender] = 0;
        (bool success, ) =
msg.sender.call.value(refund)("");
        require(success);
    }
}
```

2) The DoS attack was launched due to exceeding the gas limit of the block. A block gas limit was implemented to prevent an attacker from creating an infinite loop in transactions. If the gas usage for a transaction exceeds this limit, the transaction will be blocked. An example of such a vulnerability is the execution of logic in a loop with a very large number of iterations. Therefore, even if there is no malicious attack, the smart contract can have problems due to exceeding the gas limit. For example, a large set of users to send funds to can exceed the gas limit and prevent a successful transaction, potentially blocking funds forever. The following code snippet displays this situation:

```
struct Payee {
    address addr;
    uint256 value;
}

Payee[] payees;

function payOut() {
    while (i < payees.length) {

payees[i].addr.send(payees[i].value);
      i++;
    }
}
```

A more serious problem is when an attacker manipulates the amount of gas used by the contract so that it reaches the limit and the transaction process fails.

To avoid this vulnerability, it is better to split such a transaction into several and check in the pre-condition of the cycle whether the gas limit has been exceeded. The secure code is described in the following fragment:

```
struct Payee {
    address addr;
    uint256 value;
}

Payee[] payees;
uint256 nextPayeeIndex;

function payOut() {
    uint256 i = nextPayeeIndex;
    // check that the amount of used
gas
    // does not exceed the limit for
the block
    while (i < payees.length &&
msg.gas > 200000) {

payees[i].addr.send(payees[i].value);
      i++;
    }
    nextPayeeIndex = i;
}
```

6. *Type mismatch vulnerability*. Solidity is a strongly typed programming language that can automatically check if a program has a type mismatch. For example, if a string value is assigned to an integer variable, a type mismatch error will occur. However, in smart contracts, even if the type does not match in some cases, the contract cannot throw an exception at runtime. The lack of manual audit of a smart contract can lead to a type mismatch vulnerability.

7. *Vulnerability of unknown function call*. Like most programming languages, a smart contract ensures the uniqueness of functions by matching the function name and number of parameters. If the function name and the number of parameters do not match any function in the called contract, then the fallback function is automatically invoked. Therefore, unexpected security problems can arise if malicious code is hidden in the fallback function.

8. *Vulnerability of an unprotected call of the self-destruct function of a smart contract*. Smart contracts can use the *delegatecall* function to call functions in an external contract. A function called using *delegatecall* is executed in the context of the contract that called it. Therefore, if the called external function has self-destruct operations such as *selfdestruct* or *suicide*, the Ether in the current contract (which calls the function via *delegatecall*) will likely be frozen forever due to the execution of

such a self-destruct operation. The following code snippet demonstrates two smart contracts: a *MaliciousContract* created by an attacker and a *VulnerableContract* that has a vulnerability of an unprotected call of the self-destruct function:

```
contract MaliciousContract {
    function getMoney() {
  // destroy the contract
VulnerableContract

selfdestruct(payable(address(this)));
    }
}

contract VulnerableContract {
    function call(address a) {
  // call the getMoney method of the
MaliciousContract contract

a.delegatecall(bytes4(sha3("getMoney()
")));
    }
}
```

If the *VulnerableContract* calls the *getMoney* method of an external *MaliciousContract* through the *call* function, the *VulnerableContract* contract will self-destruct, because the *selfdestruct* method in the *getMoney* method will be called in the context of the *VulnerableContract* contract.

The *delegatecall* method should be used for contracts that are fully trusted.

This attack was used in the Parity protocol. An anonymous user found and exploited the vulnerability in the library's smart contract, making himself the owner of the contract. The attacker then proceeded to self-destruct the contract. This resulted in the blocking of funds in 587 unique wallets with a total amount of 513,774.16 Ether.

9. *Exploitation of the transaction source.* The Ethereum smart contract has a global variable, namely *tx.origin*, which can track the entire call stack and return the address that initiated the transaction. If the contract uses this global variable for authorization and authentication, attackers can use this feature of *tx.origin* to develop a suitable attack to steal Ether. For example, another contract can use the fallback function to re-call the smart contract and pass authorization. The following code snippet demonstrates a vulnerable smart contract, *VulnerableContract*, which writes the address of the contract that created it to the *owner* variable during creation:

```
contract VulnerableContract {
    address owner;

    constructor() public {
        owner = msg.sender;
    }

    function transferTo(address
payable dest, uint amount) public {
        require(tx.origin == owner);
        dest.transfer(amount);
    }
}
```

When the *transferTo* method is called, authorization is first performed by comparing the value of *tx.origin* and the *owner* variable, and then the Ether is transferred to the attackers' smart contract address.

The following snippet demonstrates the *MaliciousContract* code*:*

```
interface TxUserWallet {
    function transferTo(address
payable dest, uint amount) external;
}

contract MaliciousContract {
    address payable owner;

    constructor() public {
        owner = msg.sender;
    }

    function() external {

TxUserWallet(msg.sender).transferTo(ow
ner, msg.sender.balance);
    }
}
```

When the *VulnerableContract* sends Ether to this contract, it calls a callback function that calls the vulnerable smart contract's *transferTo* method again. This way, when the *transferTo* method is executed, the value of *tx.origin* will remain the past, meaning the value will be the address of the *VulnerableContract* contract, not the *MaliciousContract* contract. Thus, attackers will be able to pass authorizations and transfer Ether from the *VulnerableContract* until it ends.

Developers should use the *msg.sender* global variable instead of *tx.origin* to authorize users.

10. *Strict balance comparison.* Strict balance comparison means that the contract execution logic relies on the contract balance being equal to an exact value, but the value of the contract balance is transparent and can be changed by any user. For example, when a contract calls the self-destruct

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

function, Ether can be sent to any contract and the receiving contract cannot reject the transaction. If the contract has a vulnerability such as strict balance comparison, an attacker can change the contract balance and cause the contract execution logic to fail. As shown in the following code snippet, it is safer to use the "greater than or equal" comparison operator to compare *this.balance* to *3ether* rather than the "strictly equal" comparison:

```
if (this.balance == 3ether): // bad
approach
dosomething();
if (this.balance >= 3ether): // good
approach
dosomething();
```

6. CONCLUSIONS

Smart contracts have become widespread thanks to the emergence of blockchains that support them, including Ethereum. The scope of smart contracts application is quite wide: they are used by cryptocurrency exchanges, DeFi and Game-Fi projects, and various platforms on the blockchain, from NFT markets to metauniverses and real estate trading.

This article describes the blockchain architecture, provides an overview of smart contracts on the Ethereum blockchain platform, and identifies the existing types of smart contracts. The life cycle of smart contracts is analyzed in detail.

The requirements for smart contracts were analyzed depending on the subject area of their application, the features of the requirements for such smart contracts were determined. For the first time, the terms "measure of coincidence of requirements" and "measure of uniqueness of requirements" were proposed using the corresponding functions for different subject areas. Measures of coincidence and uniqueness of requirements for nine subject areas were determined. It is concluded that it is necessary to determine the mechanisms and templates for collecting requirements, taking into account the characteristics of the subject area. The proposed measures will make it possible to obtain a quantitative assessment of templates for collecting requirements for smart contracts and other types of software in the future.

The article considers 10 types of vulnerabilities of smart contracts at the Solidity code level: reentrancy vulnerability, integer overflow/underflow vulnerability, access control vulnerability, vulnerability of improperly handled exception from an external function, denial of service vulnerability, type mismatch vulnerability, vulnerability of unknown function call, vulnerability of an unprotected call of the self-destruct function of a smart contract, exploitation of the transaction source, strict balance comparison. Approaches to avoid these vulnerabilities and examples of their exploitation by attackers are presented.

REFERENCES

1. Janssen, A. & Patti, F. "Demystifying smart contracts". *Osservatorio Diritto Civile Commerciale*. 2020; 9 (1): 31–50. DOI: https://doi.org/10.4478/98131.

2. Zou, W., Lo, D., Kochhar, P. S., Le, X.-B.-D., Xia, X., Feng, Y., Chen, Z. & Xu, B. "Smart contract development: Challenges and opportunities". *IEEE Trans. Softw. Eng*. https://www.webofscience.com/wos/woscc/full-record/WOS:000707441900003. 2021; 47 (10): 2084–2106. DOI: https://doi.org/10.1109/TSE.2019.2942301.

3. Tolmach, P., Li, Y., Lin, S.-W., Liu, Y. & Li, Z. "A survey of smart contract formal specification and verification". *ACM Comput. Surv*. https://www.webofscience.com/wos/woscc/full-record/WOS:000697296500014. 2022; 54 (7): 1–38. DOI: https://doi.org/10.1145/3464421.

4. Heise, K. "DeFi lost over $47 billion in 2022: DEFIYIELD report". *Bsc news*. 2023. – Available from: https://bsc.news/post/defi-lost-over-47-billion-in-2022-defiyield-report. – [Accessed: Nov. 2022].

5. Abdellatif, T. & Brousmiche, K. L. "Formal verification of smart contracts based on users and blockchain behaviors models". *Proceedings of the IFIP NTMS. IEEE*. https://www.webofscience.com/wos/woscc/full-record/WOS:000448864200067. *Electronic*. 2018. p. 1–5. DOI: https://doi.org/10.1109/NTMS.2018.8328737.

6. Ahrendt W., Bubel, R., Ellul, J., Pace, G. J., Pardo, R., Rebiscoul, V. & Schneider, G. "Verification of smart contract business logic". *Proceedings of the FSEN. Springer International Publishing*. 2019. p. 228–243. DOI: https://doi.org/10.1007/978-3-030-31517-7_16.

7. “ETSI GS PDL 011 V1.1.1, Permissioned distributed ledger ETSI industry”. *Specication Group, ETSI*. Sophia Antipolis: France: 2021. – Available from: https://www.etsi.org/technologies/permissioned-distributed-ledgers. – [Accessed: Nov. 2022].

8. “A guide to the project management body of knowledge and the standard for project management”. *PMBOK® Guide*. Seventh Edition. USA. 2021. 274 p.

9. “A guide to the business analysis body of knowledge”. *BABOK® Guide.* – Available from: https://www.iiba.org/career-resources/a-business-analysis-professionals-foundation-for-success/babok/. – [Accessed: Nov. 2022].

10. Mazurok, I. E., Leonchyk, Y. Y., Antonenko, O. S. & Volkov, K. S. “Smart contract sharding with proof of execution”. *Applied Aspects of Information Technology*. 2021; 4 (3): 271–281. DOI: https://doi.org/10.15276/aait.03.2021.6.

11. Mazurok, I.Y., Leonchyk, Y.Y., Grybniak, S.S., Nashyvan, O.S. & Masalskyi, R.O. “An incentive system for decentralized DAG-based platforms”. *Applied Aspects of Information Technology*. 2022; 5 (3): 196–207. DOI: https://doi.org/10.15276/aait.05.2022.13.

12. Wang, H., Liu, Z., Ge, C.P., Sakurai, K. & Su, C.H. “A privacy-preserving data feed scheme for smart contracts”. *IEICE Transactions on Information and Systems*. https://www.webofscience.com/wos/woscc/full-record/WOS:000748957000002. 2022; E105D (2): 195–204. DOI: https://doi.org/10.1587/transinf.2021BCI0001.

13. Farahat, I. S., Aladrousy, W., Elhoseny, M., Elmougy, S. & Tolba, A. E. “Improving healthcare applications security using blockchain”. *Electronics,* https://www.webofscience.com/wos/woscc/full-record/WOS:000887069100001. 2022; 11 (22): 3786. DOI: https://doi.org/10.3390/electronics11223786.

14. Hawashin, D., Salah, K., Jayaraman, R., Yaqoob, I. & Musamih, A. A. “Blockchain-based solution for mitigating overproduction and underconsumption of medical supplies”. *IEEE Access*. https://www.webofscience.com/wos/woscc/full-record/WOS:000838387800001. 2022; 10: 71669–71682. DOI: https://doi.org/10.1109/ACCESS.2022.3188778.

15. Sun, H., Wang, X. Y. & Wang, X. E. “Application of blockchain technology in online education”. *International Journal of Emerging Technologies in Learning,* https://www.webofscience.com/wos/woscc/full-record/WOS:000448443800019. 2018; 13 (10): 252–259. DOI: https://doi.org/10.3991/ijet.v13i10.9455.

16. Awaji, B. & Solaiman, E. “Design, implementation, and evaluation of blockchain-based trusted achievement record system for students in higher education”. *CSEDU: Proceedings of the 14th International Conference on Computer Supported Education,* https://www.webofscience.com/wos/woscc/full-record/WOS:000814755400023. 2022; 2: 225–237. DOI: https://doi.org/10.5220/0011044200003182.

17. Upadhyay, K., Dantu, R., Zaccagni, Z. & Badruddoja, S. “Is your legal contract ambiguous? Convert to a smart legal contract”, https://www.webofscience.com/wos/woscc/full-record/WOS:000647642100032. *IEEE International Conference on Blockchain*. 2020: 273–280. DOI: https://doi.org/10.1109/Blockchain50366.2020.00041.

18. Shah, A. & Alsadiey, J. “Legal adaptation of smart contracts (analytical study)”. *International Journal of Early Childhood Special Education,* https://www.webofscience.com/wos/woscc/full-record/WOS:000791906700004. 2022; 14 (3): 1802–1807. DOI: https://doi.org/10.9756/INT-JECSE/V14I3.210.

19. Meng, Q.F. & Sun, R.G. “Towards secure and efficient scientific research project management using consortium blockchain”. *Journal of Signal Processing Systems for Signal Image and Video Technology,* https://www.webofscience.com/wos/woscc/full-record/WOS:000524385500001. 2020; 93(2-3): 323–332. DOI: https://doi.org/10.1007/s11265-020-01529-y.

20. Chen Y.W. “Blockchain in enterprise: An innovative management scheme utilizing smart contract”. https://www.webofscience.com/wos/woscc/full-record/WOS:000588568400005. *9th International Conference on Industrial Technology and Management*. 2020. p. 21–24.

21. Lenarduzzi, V., Lunesu, M.I., Marchesi, M. & Tonelli, R. “Blockchain applications for agile methodologies”. *19th International Conference on Agile Software Development*

https://www.webofscience.com/wos/woscc/full-record/WOS:000474466600030. 2018. DOI: https://doi.org/10.1145/3234152.3234155.

22. "This is the very first iteration of the decentralized application security project Top 10 of 2018". *NCC Group.* – Available from: https://dasp.co/. – [Accessed: Nov. 2022].

23. "Smart contract weakness classification and test cases". *SWC Registry*. – Available from: https://swcregistry.io/. – [Accessed: Nov. 2022].

24. Kalra, S., Goel, S., Dhawan, M. & Sharma, S. "ZEUS: Analyzing safety of smart contracts". *25th Annual Network and Distributed System Security Symposium*. https://www.webofscience.com/wos/woscc/full-record/WOS:000722005800065. 2018. DOI: https://doi.org/10.14722/ndss.2018.23082.

25. Gao, J.B., Liu, H., Liu, C., Li, QSLi., Guan, Z. & Chen, Z. "EASYFLOW: Keep ethereum away from overflow". *IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings*. https://www.webofscience.com/wos/woscc/full-record/WOS:000503272600010. 2019. p. 482–489. DOI: https://doi.org/10.1109/ICSE-Companion.2019.00029.

26. Akca, S., Rajan, A. & Peng, C. "SolAnalyser: A framework for analysing and testing smart contracts". *26th Asia-Pacific Software Engineering Conference (APSEC)*. https://www.webofscience.com/wos/woscc/full-record/WOS:000517102200060. 2019. p. 482–489. DOI: https://doi.org/10.1109/APSEC48747.2019.00071.

27. "Three types of smart contracts. How to develop a smart contract?". *Merehead.* – Available from: https://merehead.com/blog/develop-smart-contract/. – [Accessed: Nov. 2022].

28. Gavin, W. "Ethereum: A secure decentralised generalised transaction ledger". *Ethereum Yellow Book.* Berlin. 2022. p. 41–42. – Available from: http://surl.li/glboa. – [Accessed: Nov. 2022].

29. Liu, C., Liu, H., Cao, Z., Chen, Z., Chen, B. & Roscoe, B. "ReGuard: Finding reentrancy bugs in smart contracts". *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. Association for computing machinery. New York. 2018. p. 65–68.

30. "Ethereum smart contract best practices". *Consensys.* – Available from: https://consensys.github.io/smart-contract-best-practices/attacks/reentrancy/. – [Accessed: Nov. 2022].

31. "Denial of service (DoS) attack on smart contracts". *Finxter.* – Available from: https://blog.finxter.com/denial-of-service-dos-attack-on-smart-contracts/. – [Accessed: Nov. 2022].

Conflicts of Interest: the authors declare no conflict of interest

Received 23.12.2022
Received after revision 18.03.2023
Accepted 25.03.2023

DOI: https://doi.org/10.15276/hait.06.2023.4
УДК 004.4'24

Вимоги до розробки смарт-контрактів та огляд вразливостей смарт-контрактів на рівні Solidity-коду на платформі Ethereum

Комлева Наталія Олегівна[1)]
ORCID: http://orcid.org/0000-0001-9627-8530; komleva@op.edu.ua. Scopus Author ID: 57191858904
Терещенко Олександр Ігорович[1)]
ORCID: http://orcid.org/0000-0003-4510-5255; alexandr.tereschenko2014@gmail.com. Scopus Author ID: 57705566400
1) Національний університет «Одеська політехніка», пр. Шевченка, 1. Одеса, 65044,Україна

АНОТАЦІЯ

Стаття присвячена розгляду автоматизованих децентралізованих програм на блокчейні, які є сучасним інструментом обробки транзакцій без допомоги довіреної третьої сторони. Метою дослідження є узагальнення та систематизація інформа-

ції щодо вимог, які висуваються до смарт-контрактів, а також огляд вразливостей смарт-контрактів на рівні Solidity-коду. Вивчено архітектуру блокчейну та визначено переваги смарт-контрактів у порівнянні зі звичайними контрактами, а саме: зменшення ризиків, скорочення витрат на адміністрування та обслуговування та підвищення ефективності бізнес-процесів. Проведено ретельний аналіз сучасних літературних джерел та виявлено поточні проблеми, з якими стикаються користувачі та розробники смарт-контрактів. Зазначено, що процес розробки смарт-контрактів є недостатньо стандартизованим та доцільним є створення системи рекомендованих вимог для смарт-контрактів, використовуваних у різних предметних областях. Зібрано та проаналізовано вимоги до смарт-контрактів для областей, що стосуються медичної сфери, області навчання, бізнесу, управління проектами, аналізу даних, розробки програмного забезпечення, організації торгівлі, логістики та юридичної сфери. Визначено, що обов'язковими вимогами для всіх цих предметних областей є безпека, прозорість процесів, визначення умов та критеріїв успіху роботи та автоматизація роботи. Проаналізовано решту вимог та введено поняття міри співпадіння та унікальності вимог для певної предметної області, які спирається на відповідні функції. Для розглянутих предметних областей обчислено міри співпадіння та унікальності. Запропоновані міри дозволять у подальшому отримувати кількісну оцінку шаблонів для збору вимог до програм з урахуванням використовуваної предметної області. Проведено огляд та систематизовано види вразливостей смарт-контрактів на рівні Solidity-коду на платформі Ethereum. Визначено найкращі практики, які дозволяють уникнути подібних вразливостей, та можливі приклади їх експлуатації з боку зловмисників. Показано, що підвищення надійності смарт-контрактів посприяє збільшенню довіри до блокчейну серед користувачів.

Ключові слова: Блокчейн; смарт-контракт; вимога; міра співпадіння; міра унікальності; Ethereum; вразливість; транзакція

ABOUT THE AUTHORS

Nataliia O. Komleva - PhD (Eng), Associate Professor, Head of Software Engineering Department, Odessa National Polytechnic University, 1, Shevchenko Ave. Odessa, 65044, Ukraine

ORCID: http://orcid.org/0000-0001-9627-8530; komleva@opu.ua**.** Scopus Author ID: 57191858904

Research field: Data analysis; software engineering; knowledge management

Комлева Наталія Олегівна – канд. техніч. наук, завідувач кафедри Інженерії програмного забезпечення Національного університету «Одеська політехніка», пр. Шевченка,1, Одеса, 65044, Україна

Oleksandr I. Tereshchenko - Master of Science, graduate student, Software Engineering Department, Odessa National Polytechnic University, 1, Shevchenko Ave. Odessa, 65044, Ukraine
ORCID: http://orcid.org/0000-0003-4510-5255; alexandr.tereschenko2014@gmail.com. Scopus Author ID: 57705566400
Research field: Machine learning; blockchain

Терещенко Олександр Ігорович – аспірант кафедри Інженерії програмного забезпечення. Національний університет «Одеська політехніка», пр. Шевченка, 1. Одеса, 65044, Україна