UDC 004.43
Volodymyr Peschanenko, Viktor Rud
Kherson State University, Kherson, Ukraine
ORCID ID 0000-0003-1013-9877
ORCID ID 0009-0009-9388-2041

OVERVIEW OF VULNERABILITIES IN SMART CONTRACTS WRITTEN SOLIDITY

DOI 10.14308/ite000787

Smart contracts (SCs), written in the programming language Solidity, are executed on the Ethereum Virtual Machine – a remote distributed system of virtual machines for executing code. The state and transaction history of SCs are stored on the blockchain. SCs open up new opportunities for the automation and decentralization of various fields: finance (DeFi), gaming (GameFi), legal, administrative, and educational. The growing popularity of blockchain technologies and the active use of SCs highlight the importance of addressing their security.

The main objectives of this article are to review scientific research and analyze significant vulnerabilities of SCs. The review of scientific research provides an objective assessment of the current state of knowledge in the field of SC security. The analysis of vulnerabilities is necessary for a brief introduction to known vulnerabilities and an understanding of their consequences.

For the research, the following scientific methods were used: literature analysis, vulnerability classification, analysis of protection methods, and review of real attacks. Analyzing historical cases of attacks on SCs creates a foundation for building theoretical attack models. Theoretical attack models are essential for a deep understanding of the nature of vulnerabilities. Once the essence of an attack is understood, recommendations for protecting SCs are formed, including libraries, secure templates, and methods of protection. Furthermore, studying historical cases of attacks and measures taken to minimize or eliminate them is useful for quick response to future similar attacks.

In the context of increasing blockchain technology complexity, checking SCs for vulnerabilities becomes of paramount importance. This study concludes the significance of developing SC security, systematically updating knowledge, and periodically rechecking contracts.

This article may be of interest to those involved in SC development and cybersecurity research.

Keywords: *blockchain, Ethereum, Solidity, smart contract, vulnerabilities, cybersecurity, cyberattack*

Introduction

Solidity [1] is the primary programming language for creating smart contracts (SC) on Ethereum [2] and Ethereum-compatible blockchains. Solidity, being Turing-complete, offers extensive development capabilities but also requires particular attention to security due to potential vulnerabilities. To date, new types of vulnerabilities in SCs continue to be discovered periodically. The discovery of a new vulnerability in one contract often leads to the identification of similar vulnerabilities in other contracts created earlier. Therefore, it is crucial to continuously monitor the security landscape of SCs.

In recent years, blockchain technology, and in particular, SCs written in Solidity, have become increasingly popular. They are being applied in various fields such as finance, law, management, gaming, and education. Let's consider a few examples of decentralized applications (DApp) that utilize SCs:

- Layer3 [3]: Layer3 is an educational platform that offers theory and practice for immersion in Web3 in a gamified format. Web3 is a decentralized internet based on blockchain technologies, allowing users to interact directly with applications and services without intermediaries.
- MetaMask (MM) [4]: Primarily a cryptocurrency wallet and a tool for interacting with DApps. It allows users to securely store and manage assets. In addition to its primary function as a cryptocurrency wallet, MetaMask (MM) also offers essential decentralized finance (DeFi) application features: swap (exchange of assets within the same network), bridge (transfer of assets between different networks), and stake (investment of assets to earn interest).
- Binaryx [5]: Binaryx is a DApp for fractional ownership of tokenized real estate.

SCs offer advantages such as transparency and immutability of transaction history, and automated verification of agreement conditions. As their application expands, security concerns become more prominent. Vulnerabilities in SCs can lead to significant financial and reputational losses for DApps.

The main objective of this article is to provide a brief overview and analysis of several important SC vulnerabilities and to review research in the field of SC vulnerabilities to understand the current state of affairs. The issue of SC vulnerabilities in Solidity has both scientific and practical significance. On one hand, it necessitates deeper research into security mechanisms, and on the other, it emphasizes the need for developing reliable approaches and standards. Maintaining high levels of SC security is crucial for the expansion of blockchain technology.

The structure of the article is as follows. The initial part contains descriptions of the methodology and the basics of Solidity. This is followed by a review of scientific articles and important vulnerabilities. The article concludes with a summary and outlines plans for further research.

1 Research Methods

To achieve the research objectives, the following methods and approaches were used:

- Literature Review: To assess the existing knowledge on the topic of SCs security, an analysis of five scientific articles dedicated to this topic was conducted. The main stages of the analysis included:
  - Selection of relevant scientific works published in peer-reviewed journals and conferences.
  - Evaluation of the significance of each work and its contribution to the field of SCs security.
  - Synthesis of research results to identify major trends and vulnerabilities.
- Classification of Vulnerabilities: The study involved classifying the main types of vulnerabilities in SCs and determining their relative severity.
- Analysis of Protection Methods: A brief analysis of existing methods for protecting SCs was conducted.

Review of Real Attacks: To gain a deeper understanding of vulnerabilities, an analysis of real attack cases was carried out.

2 Basics of Smart Contracts and the Solidity Programming Language

In the context of this article, understanding the basics of Solidity and SCs is essential for comprehending how vulnerabilities can arise and how they can be prevented.

2.1 Basics of Smart Contracts

Smart contracts are self-executing contracts that contain the terms of the agreement between the parties involved in the transaction. The code and states of SCs (variable values) are stored on the blockchain, which provides additional transparency if needed, as the state variables can be viewed. State variables are variables that retain their value between function calls in the SC and are stored on the blockchain. If the predefined conditions of the transaction are met, the execution of function or their sequence occurs successfully. This is achieved through the combination of control statements (programming constructs that manage the flow of program execution). Thus, SCs allow for reliable agreements without the need for third-party involvement.

SCs also ensure the transparency of operations, reducing time and material costs. When using SCs, the possibility of non-compliance with the terms of the agreement is eliminated. Therefore, SCs automate the execution of contracts.

2.2 Key Characteristics and Principles of Solidity

The characteristics of Solidity provide a general understanding of the Solidity programming language. The principles of Solidity guide the correct way to write code.

Characteristics of Solidity. The specific characteristics of Solidity make it a powerful tool for creating reliable and functional contracts on the Ethereum network. These characteristics include the following elements:

- Static Typing: The data type of each variable is specified at the time of its declaration, which helps prevent errors related to incorrect typing.
- Inheritance and Interfaces: Inheritance allows the creation of complex contracts with hierarchy and shared methods. Interfaces enable interaction with contracts without knowing their internal structure. These constructs are similar to classical object-oriented programming (OOP).
- State Management: Contracts can store and modify their states, preserving data between transactions.
- Functions and Modifiers: Functions control the behavior of the contract. While modifiers can contain any code, they are usually used to control the flow of execution. In Solidity, there cannot be multiple parallel execution threads, only one.
- SCs and Transactions: Contracts respond to transactions. They can send and receive Ether (the coin used to pay for gas in the Ethereum network), interact with other contracts, and execute functions.

Error Handling: The error management system allows handling exceptional situations in contracts. This enables quick identification of the problematic area. Some of the most common errors are standardized [6].

Principles of Solidity. The principles of Solidity guide developers in the process of creating efficient and reliable SCs. They encompass the following ideas and approaches that underpin Solidity:

- Contract-oriented approach: Each SC in Solidity is considered an autonomous unit with its own state and behavior. An undeployed contract can be viewed as analogous to a class in OOP, while a deployed contract is analogous to an object.
- Security and predictability: Principles ensuring the reliability and stability of the code, including strict data typing and exception handling.
- Determinism: Code must execute in a consistent and predictable manner across all nodes in the Ethereum network.

- Gas efficiency: Efficient use of network resources and minimizing gas costs for operations. Gas in Ethereum is a unit of measurement for computational effort. It has a variable cost and is paid in Ether.

Standards compliance: Adhering to established standards for compatibility when interacting within the Ethereum ecosystem [6].

3 Analysis of Research on Smart Contract Vulnerabilities

To understand the current needs and trends in the security of SCs written in Solidity, several articles were selected and reviewed.

3.1 Formalization of Solidity Syntax and Semantics

The article [7] explores the formalization of syntax and semantics in Solidity to study the language's features. Syntax refers to the rules and structure that define how instructions in a programming language should be composed. Semantics refers to the meaning and behavior of those instructions. The article describes the formal syntax and semantics for Lolisa, a subset of the Solidity programming language. Lolisa includes nearly all components of Solidity's syntax and features an enhanced static typing system to improve type safety. This research is important for understanding the deeper aspects of Solidity, including its type system and code organization. It can contribute to the development of more secure and reliable SCs.

3.2 Recommendations for Improving Smart Contract Security

The study cited in [8] focuses on evaluating the security of SCs. The authors examine various aspects of security and propose methods to enhance SC protection. The work emphasizes the importance of thorough analysis and testing of SCs before deployment, considering the high risks associated with financial and other sensitive operations. The study classifies 33 Solidity vulnerabilities according to the universal taxonomy of the Common Weakness Enumeration (CWE) [9]. The authors conduct a comparative analysis of the capabilities of various static analysis tools for detecting vulnerabilities in SCs and suggest combining these tools to improve vulnerability coverage. Ultimately, the study provides recommendations for enhancing SC security by identifying the most critical vulnerabilities not covered by static analysis tools (a method for analyzing program source code without executing it).

3.3 Meta-Analysis of Vulnerability Studies

The study cited in [10] is a systematic review of scientific research and publications from 2016 to 2021, dedicated to SC technology and its security. The primary goal of the work is to identify insufficiently studied vulnerabilities and methods for protecting SCs. The authors also list all datasets used by previous researchers, which can help create more effective machine learning models in the future. Additionally, a comparison of static analysis tools for SCs is conducted, taking into account various characteristics.

This study contributes to the understanding of SC vulnerabilities in the following ways:

- Systematic Review: The article provides a comprehensive analysis of scientific works, offering an extensive overview of the current state of research on SC vulnerabilities.
- Identification of Research Gaps: Attention is given to underexplored aspects of vulnerabilities, which can guide future research to fill these gaps.

Comparison of Analysis Tools: The comparison of various static analysis tools for SCs offers valuable insights into their effectiveness and areas of application.

3.4 Metadata Collection for Vulnerability Research

The study cited in [11] describes an automated method for extracting and classifying vulnerabilities in Ethereum SCs, as well as corresponding fixes. These data are extracted from GitHub repositories and vulnerability records in the Common Vulnerabilities and Exposures [12], which are stored in the National Vulnerability Database [13].

The developed method is implemented in a fully automated system called AutoMESC, which uses seven of the most well-known SC security tools to classify and label the collected vulnerabilities.

AutoMESC collects metadata that can be used in data-driven SC security research (e.g., vulnerability detection, vulnerability classification, severity prediction, and automated repair). Currently, the dataset contains 6,700 "vulnerability-fix" pairs for SCs in Solidity. The quality of the dataset is evaluated in terms of accuracy, provenance, and timeliness. These data can be used to improve methods for vulnerability detection, classification, and the development of strategies for their mitigation, thereby enhancing the overall security of SCs.

3.5 Actual Exploitation of Smart Contract Vulnerabilities

The study cited in [14] concludes, after analyzing vulnerable contracts, that the actual exploitation of these vulnerabilities occurs quite rarely, despite the large overall number of vulnerable SCs. This highlights several important points:

Risk Reevaluation: Although many SCs are technically vulnerable, the real risk of their exploitation may be significantly lower than expected.

Updating vs. Deletion: In the context of blockchain, especially in systems like Ethereum, SCs are generally not deleted but rather updated or replaced with new versions. Old versions of contracts remain on the blockchain permanently, as the blockchain is an immutable ledger. Such old contracts may no longer be used as users transition to new versions. However, in some cases, old contracts may contain user assets that have not transitioned to the new version. Nonetheless, these unused contracts often do not contain significant funds or assets, reducing the incentive for attackers to exploit them. Therefore, even if old versions of contracts remain technically vulnerable, the real risk of their exploitation is minimal.

4 Brief Overview of Common Vulnerabilities and Issues

In this section, the main types of contract vulnerabilities are briefly reviewed along with their historical examples.

4.1 Reentrancy Attacks

Reentrancy attacks [15 – 16] in Solidity occur when a victim contract, instead of interacting with an expected Externally Owned Account (user address), interacts with a contract address (CA). The CA may contain malicious code in its fallback methods. Fallback methods are functions in SCs that are automatically invoked when the contract receives Ether or data that does not match any other function. When the victim contract calls the attacking contract, the execution flow is transferred to the called contract. At this point, the malicious code gets executed. By using "sub-calls" within the main call, the attacking contract can repeatedly call the vulnerable function to drain the attacked contract or otherwise manipulate the contract's data for gain. This can be repeated as many times as the attacker needs. For example, if an SC sends funds before updating its internal state, the attacker can repeatedly call this function.

Causes:

- Nested Calls: Reentrancy attacks are possible when an SC function calls an external contract, allowing it to perform actions that can re-enter the vulnerable function of the original contract.
- State Changes After External Calls: The vulnerability arises if critical state changes (e.g., balance updates) occur after the external call.

To protect against this, one or a combination of developed protection methods should be used, including:

Using the Checks-Effects-Interactions (CEI) Pattern: A pattern that clearly defines the sequence of actions. First, the conditions under which the function should be executed are

checked. Second, the contract's state is changed. Only after these steps does the interaction with external contracts occur.

Applying Mutexes: A mutex is a switch at the beginning of a function. It registers entry into the function by changing the state variable and prevents re-entry within the same call. Although it can be implemented differently, its main principle remains unchanged.

Historical Example. The DAO hack [17] in 2016 is a striking example of a reentrancy attack. The DAO was a complex system of contracts designed to operate as a venture fund without human intervention. An attacker discovered a reentrancy vulnerability in the withdrawal function. This function did not update the internal balance before sending Ether, allowing reentrancy calls. The CEI pattern was not followed. The attacker created a malicious contract that initiated the withdrawal of funds from the DAO. During the withdrawal, the malicious contract repeatedly called the withdrawal function several times before the original transaction was completed and the balance updated. This allowed the attacker to withdraw Ether from the DAO multiple times, leading to significant loss of funds and serious debates within the Ethereum community. Ultimately, the hack led to a controversial hard fork of the Ethereum blockchain to return the stolen funds. A hard fork is a significant update to the blockchain protocol that is not compatible with previous versions, leading to the blockchain splitting into two chains. As a result of the DAO incident, two separate chains were created: Ethereum (ETH) and Ethereum Classic (ETC). In the ETH version, the stolen funds were returned to the victims, while in the ETC version, they remained with the attacker.

This incident underscored the importance of security in SC development and prompted the blockchain community to pay more attention to security practices and audits. This subtype of reentrancy, called Single-Function Reentrancy, now seems trivial. However, in recent years, four more subtypes of Reentrancy Attack have been identified based on it [18]: cross-chain reentrancy, cross-function reentrancy, read-only reentrancy, and cross-contract reentrancy.

4.2 Integer Overflows and Underflows

Issues with handling integer operations before compiler version 0.8 could lead to incorrect calculations and vulnerabilities. The compiler in Solidity is a program that converts source code written in the Solidity programming language into bytecode executable by the Ethereum Virtual Machine (EVM) [19] — a remote distributed system of virtual machines for code execution. Integer overflows and underflows were previously common vulnerabilities [20]. These vulnerabilities arose due to arithmetic operations and data type limitations. Variables could take on values outside the allowable range, resulting in overflows or underflows. These errors were often caused by insufficient input validation and improper range management. Let's look at the basic definitions:

- Integer overflow: Occurs when a variable reaches its maximum value and rolls over to its minimum value (for example, 255 + 1 becomes 0 in an 8-bit unsigned integer type `uint8`).
- Integer underflow: Occurs when a variable drops below its minimum value and rolls over to its maximum value (for example, 0 – 1 becomes 255 in an 8-bit unsigned integer type `uint8`).

In Solidity, these vulnerabilities can lead to unexpected contract behavior, including fund leaks or logic breaches. To prevent potential issues, developers should use overflow and underflow checks and safe math libraries, such as Math from OpenZeppelin [21].

In newer versions of Solidity, the issue of integer overflows and underflows has been mitigated by the introduction of built-in checks by default. If detected, the transaction is reverted: all changes are rolled back, and unused gas is refunded to the caller. This is similar to the action of the revert() global function. Global function is a function declared in a programming language and available from any place in the code. It should be noted that such built-in checks use additional gas, and in some cases, it may be advisable to disable them for

efficiency. This can be done using the unchecked{} keyword, within which code is not subject to automatic checking.

In general, an attack using the integer overflow vulnerability can be carried out as follows:

- Preparation: The attacker analyzes the SC code for vulnerabilities related to integer overflows or underflows. The attacker may use their own SC to interact with the vulnerable contract and initiate the attack.
- Executing the Attack: The attacker sends transactions that exploit the vulnerability, such as calling a function with parameters that cause an overflow.
- Result: As a result of the attack, the attacker may obtain an unjustifiably large number of tokens or other privileges in the contract.

Historical Example. One instance of an attack related to the integer overflow vulnerability occurred with the BEC token on Ethereum in 2018 [22]. The attacker used the `batchTransfer()` function of the BEC token, where due to integer overflow, the number of tokens being transferred became incredibly large. This was caused by a vulnerable function code. The code allowed a multiplication operation whose result could exceed the maximum value for the `uint` data type. As a result, the attacker was able to withdraw a massive number of tokens, leading to significant financial losses.

Let's briefly examine the vulnerable function and identify the reasons that enabled the attack (Fig. 1).

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.25;
3  contract Test {
4    mapping(address => uint8) public balances;
5    constructor() {
6      balances[msg.sender] = 255;
7    }
8
9    // Send 128 tokens to [0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2, 0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db]
10   function batchTransfer(address[] memory _receivers, uint8 _value) public {
11     unchecked{
12       uint cnt = _receivers.length;
13       uint8 amount = uint8(cnt) * _value;
14       require(cnt > 0 && cnt <= 20);
15       require(_value > 0 && balances[msg.sender] >= amount);
16       balances[msg.sender] = balances[msg.sender] - amount;
17       for (uint8 i = 0; i < cnt; i++) {
18         balances[_receivers[i]] = balances[_receivers[i]] + _value;
19       }
20     }
21   }
22 }
```

Fig. 1. Theoretical model of a contract vulnerable to overflow

The multiplication result (line 13) may exceed the maximum value for a `uint8` variable (an unsigned integer occupying one byte). In line 15, the Checks stage includes a condition check using `require()`, which verifies the sender's balance (`balances[msg.sender]`). The balance must be greater than the transferred amount (`amount`). `require()` is a built-in function that checks a condition and reverts the transaction if the condition is not met. A built-in function is one available without definition or import. Since `amount` becomes less than the actual transferred value after the overflow in line 13, the check no longer functions correctly.

Readers are encouraged to test by sending 128 tokens to two addresses. The test result will show no spending by the sender and an addition of 128 tokens to the recipient due to the `uint8` overflow.

4.3 Unprotected Functions

Unprotected functions [23] are functions without proper verification of the caller's identity. Unprotected functions can be called by any user in the network because they lack access restrictions. This behavior allows various addresses to perform actions that were not intended to be accessible to everyone. Vulnerability arises if critical functions of the contract, such as changing ownership, updating critical variables, or transferring funds, are accessible without user rights verification.

To mitigate this risk, developers use access control mechanisms. Establishing a robust access control system is a key aspect of ensuring SCs security. For example, it is common to use various versions of modifiers that check if the caller's address matches the approved address for invoking the function. The Solmate library has an `authorities` folder with high-quality access control implementations. Additionally, the OpenZeppelin library's `access` folder can be used for access control. These implementations are usually more standardized and reliable than custom-written ones, but it is still recommended to thoroughly review them before use.

Historical Example. A well-known case describes how the trading bot 0xbad [24 – 25] with an unprotected function was hacked. The attacker discovered that the dYdX flash loan function (a decentralized platform for cryptocurrency trading) used by the bot allowed arbitrary code execution. This meant that the execution flow passed to the attacker, who could then perform the necessary actions. Once the flow returned, the vulnerable function did not have sufficient checks to prevent unauthorized actions. This allowed the attacker to withdraw all the bot's funds (1101 ETH) to their own address. Using an access modifier could have prevented the execution flow from being passed to an unknown address.

4.4 Short Address or Parameter

Short address or parameter vulnerability [26] is related to the incorrect handling of addresses and parameters, leading to the misinterpretation of data. It occurs when the transmitted length of an address or parameter in a transaction is shorter than expected. This causes the contract to misread the data, resulting in undesired function behavior. This attack exploits how the EVM handles data, where incomplete data can be padded with zeros on the right, potentially altering the intended address or value. Preventing such attacks requires thorough verification and normalization of input data in the contract.

This vulnerability can be dangerous for several reasons:

Unintentional Transfers: Erroneous transfers to incorrect addresses, which can lead to the loss of funds.

Distortion of Transfer Amounts: This can lead to unexpected financial losses.

Impact on Other Operations: The potential for more complex influences on other transaction parameters due to data shifting.

Example. By refining the theoretical model [27], we can illustrate the vulnerability. In the adapted theoretical model, shortened data is used for simplicity and clarity: the function expects to receive an address of 2 bytes, but receives an address of one byte. User `A` wants to transfer 64 tokens to user `B` at the address `0xab00`. If the function receives an address one byte shorter than expected, the missing bytes are added to the right of the hexadecimal encoded byte value. Thus, the encoded parameters will contain the incorrect value `ab001000` instead of `ab00100`. Encoding is the process of transforming data from one format to another, more suitable for processing or storage. As a result, 65536 tokens (10000 in hexadecimal) will be transferred instead of the intended 64 tokens (100 in hexadecimal) to the wrong address.

This vulnerability highlighted the importance of thoroughly validating input data in SCs. The vulnerability was fixed at the compiler level in version 0.5. Now, transactions are rejected if a function argument is shorter than expected.

4.5 Block Values as a Proxy for Time

The timestamp dependency vulnerability [28 – 32] occurs when critical contract functions rely on `block.timestamp` value. The `timestamp` is a property of the global variable `block` that returns the time at which the block containing the function execution was finalized. Miners can exert minor influence over `block.timestamp` value. Although their influence is limited by blockchain protocols that prevent significant deviations from real time, even this restricted ability to manipulate timestamps makes contracts that use timestamps for crucial operations vulnerable. For instance, using `block.timestamp` for random number generation or determining event timing can be compromised by miners manipulating the `block.timestamp`.

It is recommended to use alternative, more reliable mechanisms for time-dependent functions and random number generation. Generally, oracles — services that provide SCs with external data — are used for such purposes. They can provide time data and random numbers, among other things.

Example. An example of a vulnerable contract and an attack algorithm is described on the Solidity by Example resource here [33]. This is a lottery where winning depends on the block's timestamp being divisible by 15. A powerful miner sets the block's timestamp to a multiple of 15 and mines it. Once the block is successfully added to the chain, the miner claims the prize.

Although Ethereum has fully transitioned from Proof-of-Work (PoW) to Proof-of-Stake (PoS), the minor possibility of manipulating the block's timestamp remains. PoW is a consensus algorithm where miners solve problems to create blocks. PoS is a consensus algorithm where block validators are randomly selected with a probability proportional to their stake.

4.6 Race Conditions and Front Running

Race conditions and front running [34] can occur when multiple transactions attempt to modify the contract's state simultaneously. Race conditions arise from the indeterminate order of transaction execution, whereas front running involves intentional interference with this order for profit. This can lead to unpredictable outcomes since the order of transaction execution in the blockchain is not guaranteed. Miners (in PoW) and validators (in PoS) have some discretion in choosing the order of transactions included in a block, based on the gas fee size or their personal gain from the transaction sequence. If two transactions try to change the same data element, the result may depend on which transaction is processed first. These situations require careful state management and synchronization in SC code.

Let's consider the attack algorithm:

- Identifying vulnerability: The attacker searches for SCs where an important function relies on the order of transaction execution.
- Monitoring: The attacker monitors the memory pool to detect specific transactions that can be used for the attack. The memory pool (mempool) is a space in blockchain nodes where unconfirmed transactions awaiting inclusion in a block are stored.
- Preparation and execution: The attacker attempts to manipulate the transaction order in the blockchain so that their prepared transaction is processed at the right moment.
- Outcome: If the attack is successful, the attacker gains a profit.

To mitigate this vulnerability, it is essential to remove the incentives that encourage race conditions.

Example. An example of a contract vulnerable to a front running attack is described in the book Mastering Ethereum (Fig. 2) [35]. Additional explanation of this attack can be found in an article on Medium [36]. Let's consider the theoretical model of the contract. The contract offers a reward of 1 ETH for solving a hashed string. The attacker monitors the mempool for a transaction containing the correct answer ("Ethereum!") and, upon finding it, submits their

own transaction with a higher gas fee. This makes the attacker's transaction a priority for miners, and it is included first in the block. As a result, the attacker claims the reward. The subsequent transaction, arriving second, encounters an empty contract balance because the `solve()` function has already transferred the entire balance to the attacker.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.25;
contract FindThisHash {
  bytes32 constant public hash = 0xb5b5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44d5479b630ee0a;
  constructor() payable {}
  function solve (string memory solution) public {
    require(hash == keccak256(bytes(solution)));
    payable(msg.sender).transfer(address(this).balance);
  }
}
```

Fig. 2. Theoretical model of a contract vulnerable to a front running attack

4.7 Denial of Service (DoS)

Denial-of-service (DoS) attacks are a broad category of attacks on blockchain and SCs designed to cause failure, creating conditions where legitimate users cannot access the SC (or their system) or access is significantly hindered. Let's examine some types of DoS attacks.

Gas Limits, Loops, and Nested Calls. This section addresses issues related to the use of loops and nested calls in Solidity, which can lead to gas limit exceedance and inefficient gas consumption in the execution of SCs.

Loops. Loops in Solidity are programming constructs that allow repeating a block of code multiple times until a certain condition is met or for a specified number of iterations (repetitions). A potential problem with loops [37 – 38] is exceeding the gas limit due to inefficiency in the loop. This causes the function to terminate forcibly with an error and roll back the transaction. Each operation in an SC requires a certain amount of gas. Loops, especially with an indefinite or very large number of iterations, can consume a significant amount of gas.

Reasons for gas shortage:

- Lack of Ether: Insufficient Ether in the EOA to purchase the required amount of gas.
- Low gas limit: The manually set gas limit is less than what is needed for the transaction.
- Block gas limits: Currently, in the Ethereum blockchain, a block has a fluctuating gas maximum and is limited to 15 – 30M gas.

To prevent costly functions, it is necessary to optimize loops and use mappings. Mapping is a data type in Solidity that stores collections of key-value pairs. In the EVM architecture, mappings consume significantly less gas than loops but have less functionality, such as the inability to retrieve all values or search by values. If loops are used, it is important to optimize them as much as possible by:

- Limiting and reducing the number of iterations when searching and replacing data;
- Performing checks on values that affect the number of cycles, especially if they are provided as function arguments from external sources.

It should be noted that the problem of loops depleting the Ether balance of callers has been fully addressed today. This has been facilitated by the addition of functionality capable of automatically calculating and predicting the maximum required amount of gas for a transaction.

One of the main dangers of SCs with inefficient loops was the depletion of the sender's gas balance. If a loop in the contract consumed too much gas, it could lead to the gas limit for the transaction being exceeded, and the transaction would not be completed. In such a case,

the gas used before the transaction was interrupted would be lost. This could lead to significant financial losses for the sender and, additionally, to the absence of the function's intended result.

Nested Calls. Currently, this problem persists with nested calls. Wallet interfaces (tools for EOA to interact with the blockchain) warn users about the high cost of the transaction. However, inattentive users can lose their entire Ether balance by spending it on executing a costly function. [39]

Example. Due to the fact that these problems do not have an obvious beneficiary, historical data on these problems is quite difficult to find. Functions with nested calls that deplete the gas of the calling contract are sometimes erroneous and sometimes intentionally written. For example, they may masquerade as balance limit setting functions for the contract (`approve()` from the ERC-20 standard [40]). In modern realities, many users periodically reset `approve()` values to 0 en masse on special platforms. Mass repetitive transactions can negatively affect their attentiveness. In such a situation, a contract with a costly function may go unnoticed.

Block Stuffing. A block stuffing attack occurs when an attacker floods blockchain blocks with their own transactions. This leads to a significant increase in gas prices. The cost of sending transactions becomes prohibitively expensive for ordinary users, thereby blocking normal use of SCs and the network as a whole.

Historical Example. An example of such an attack can be seen in the 2018 attack on the Fomo3D gambling game [41]. Before the incident, Fomo3D was a popular game finance (GameFi) DApp. GameFi refers to the use of blockchain technology in gaming to enable players to earn financial rewards. The game was designed to reward the last address that purchased a "key" before the countdown timer expired. Each key purchase extended the timer. The game ended when the timer reached zero.

Simplified Attack Algorithm:

- Key Purchase by Attacker: The attacker bought a key, thus extending the timer.
- Block Stuffing: The attacker then proceeded to stuff 13 consecutive blocks. They executed multiple transactions that exhausted the gas limit of each block. With a high gas fee and occupying a significant portion of the mempool, the attacker's transactions were prioritized for inclusion in the blocks. This led to the exclusion of transactions from other players.
- Timer Expiration: As a result, no one else could buy a key before the timer expired.
- Payout to the Attacker: When the timer expired, the payout was automatically made to the attacker, as they were the last key holder.

This attack highlighted a vulnerability in Ethereum's block and gas mechanics and demonstrated how attackers can exploit these aspects to manipulate outcomes in DApps.

Greedy and Selfdestruct Contracts. Greedy and selfdestruct contracts are vulnerable contracts that permanently keep users' funds for themselves.

Greedy Contracts. In the context of Ethereum, a "greedy contract" refers to a SC that, due to coding errors or design flaws, cannot release funds from its balance. This can happen if the contract lacks the necessary instructions to send funds or if there are bugs in these instructions. As a result, the contract becomes "greedy," preventing the release of the funds it has received.

Selfdestruct Contracts. SCs often include the ability to be deactivated by the owner or trusted addresses in emergency situations, such as during attacks or technical failures that could lead to Ether leakage. However, if a contract can be forcibly destroyed by any arbitrary account using the `selfdestruct(address)` command, it is considered vulnerable.

`selfdestruct(address)` is a built-in function that removes the contract from the blockchain, transferring any remaining funds to the specified address.

As of the Cancun hard fork, the `selfdestruct(address)` command has effectively been rendered non-functional [42].

Historical Example. An example is the incident with the Parity wallet [43]. The incident occurred in 2017 due to a vulnerability in the wallet's code. A user accidentally exploited this vulnerability [44]. While examining the Parity wallet code, the user destroyed an essential part of the code, the `WalletLibrary` contract, using the `selfdestruct(address)` [45].

The `initMultiowned()` function (Fig. 3) provides the logic for setting the contract owners and the required number of owners needed to initiate the contract's destruction. The destruction was possible because the contract had not been initialized by the owners using the `initMultiowned()` function. Consequently, the user who triggered the `initMultiowned()` function became the owner of the `WalletLibrary` contract and was able to execute the destructive function.

```
function initMultiowned(address[] _owners, uint _required ) {
  if (m_numOwners > 0) throw;
  m_numOwners = _owners.length + 1;
  m_owners[1] = uint(msg.sender);
  m_ownerIndex[uint(msg.sender)] = 1;
  m_required = _required;
  /* ... */
}

function selfdestruct(address _to) {
  uint ownerIndex = m_ownerIndex[uint(msg.sender)];
  if (ownerIndex == 0) return;

  var pending = m_pending[sha3(msg.data)];
  if (pending.yetNeeded == 0) {
    pending.yetNeeded = m_required;
    pending.ownersDone = 0;
  }

  uint ownerIndexBit = 2 ** ownerIndex;
  if (pending.ownersDone & ownerIndexBit == 0) {
    if (pending.yetNeeded <= 1) selfdestruct(_to);
    else {
      pending.yetNeeded--;
      pending.ownersDone |= ownerIndexBit ;
    }
  }
}
```

Fig. 3. An example of combining functions that make the contract and its derived contracts vulnerable

The destroyed contract was used as a library (a contract with functions used in other SCs) and was critically important for the operation of the wallet and associated DApps. As a result, the funds stored in these wallets were locked and became inaccessible.

After the vulnerability was discovered, the destroyed contract was categorized as a selfdestruct contract. The DApps dependent on the destroyed contract, which could no longer withdraw funds, were categorized as greedy contracts.

Unexpected Revert. The unexpected revert [46] represents a situation where a function call fails and reverts despite the expectation of successful execution.

Examples. Let's examine theoretical models of vulnerable contracts. In the first example, when a user places a bid (`function bid()`), the function checks if the new bid (`msg.value`) exceeds the current highest bid (`highestBid`) (Fig. 4). If so, the contract returns the previous highest bid to its sender. However, if the bidding contract is designed such that it cannot receive Ether, the attempt to return funds results in a failure (via `require()`), causing the transaction to revert (`revert()` is embedded in `require()`). This means the CA can remain the auction leader since no subsequent bids can surpass it due to failed refund attempts.

```
1  // SPDX-License-Identifier: MIT
2  pragma solidity ^0.8.25;
3  contract Auction {
4    address currentLeader;
5    uint highestBid;
6    function bid() external payable {
7      require(msg.value > highestBid);
8      require(payable(currentLeader).send(highestBid));
9      currentLeader = msg.sender;
10     highestBid = msg.value;
11   }
12 }
```

Fig. 4. Theoretical model of a contract vulnerable to unexpected revert, `Auctiont`

The RefundManager contract (Fig. 5) is also vulnerable to contracts that cannot accept payments. Here, the vulnerable mechanism for sending funds to a list of addresses is written on line 9. If any address in the `refundAddresses` array cannot accept payments, the function ends with a `revert()`.

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.25;
contract RefundManager {
  address[] public refundAddresses;
  mapping(address => uint) public refunds;
  function refundAll() external {
    for(uint x; x < refundAddresses.length; x++) {
      require(payable(refundAddresses[x]).send(refunds[refundAddresses[x]]));
    }
  }
}
```

Fig. 5. Theoretical model of a contract vulnerable to unexpected revert, `RefundManager`

To address the unexpected revert problem, it is recommended to use a pull payments system instead of push [48]. In a push payment system, the payment initiator (contract) actively sends funds to the recipient. This system is used in the provided examples. In a pull payment system, the recipient initiates the fund transfer from the contract.

4.8 Summary

All the vulnerabilities described above are included in the Contract Weakness Classification (SWC) [4]. The SWC is a comprehensive registry that systematizes vulnerabilities characteristic of Ethereum SCs. This system includes descriptions of specific security issues, provides code examples to illustrate each vulnerability, and links them to the corresponding CWE elements. Although the SWC has not been updated by developers since 2020, it remains a valuable resource for developers and security professionals. The SWC helps better understand and prevent potential risks in SC development.

The table below (Table # 1) presents a comparative overview of the examined vulnerabilities. The last column represents the authors' subjective assessment of the maximum possible loss of funds and reputation for a DApp with a vulnerable contract.

Table # 1.
Comparative Table of Common Vulnerabilities

Vulnerability Type	Consequences	Loss of Funds and Reputation
Reentrancy Attacks	Loss of all funds due to multiple function calls before the first call completes	★★★★
Integer Overflows and Underflows	Incorrect calculations lead to unpredictable contract behavior	★★★★
Unprotected Functions	Unauthorized access to functions can lead to theft of funds or data manipulation	★★★★
Short Address or Parameter	Transaction errors due to short addresses and parameters	★★☆☆
Block Values as a Proxy for Time	Manipulations with block timestamps disrupt intended logic	★☆☆☆
Race Conditions and Front Running	Manipulations with transaction execution order disrupt intended logic	★☆☆☆
Gas Limits, Loops, and Nested Calls (DoS)	Exceeding gas limits, leading to failed transactions or loss of funds	★☆☆☆
Greedy and Selfdestruct Contracts (DoS)	Funds are locked in the contract.	★★★★
Block Stuffing (DoS)	Overloading the block with transactions, preventing other transactions from being included	★☆☆☆
Unexpected Revert (DoS)	Unexpected transaction rollback	★★☆☆

5. Unidentified Aspects of Smart Contract Vulnerabilities

As blockchain technology evolves and SCs become more complex, new vulnerabilities that have either not been studied before or only partially understood are emerging. In the field of SCs, several potential threats have been identified:

- New Attack Vectors: Advances in DApps technologies lead to the emergence of new attack vectors. For example, with the advent of complex DeFi applications, vulnerabilities become less noticeable as they are disguised within intricate financial mechanisms.
- System Integration and Interaction: It is important to mention that integration with external systems increases the attack surface. Vulnerabilities in external systems can be exploited to attack the contract.
- Regulation: Changes in the legal and regulatory landscape contribute to the emergence of new vulnerabilities. SCs must adapt to various legal standards, which can impact their security and reliability.

To maximize the prevention of vulnerabilities, continued research in SC security is essential. Focus should be given to:

- Developing new, more cost-effective and advanced testing methods;
- Analyzing the interaction of SCs with other technologies;
- Implementing artificial intelligence (AI) to assist in analysis;

- Conducting deeper studies of vulnerabilities;
- Fostering community collaboration for knowledge and experience sharing.

Conclusions

This article provides a brief analysis of several vulnerabilities, examining key aspects and security issues. Despite significant efforts by the developer community to improve the security of SCs, the problem remains relevant and requires further research and development.

The security of SCs is critically important for the future of blockchain technologies. SCs are fundamental to DApps. The reliability of SCs reduces financial risk and increases user and investor trust in the blockchain ecosystem.

Based on the analysis conducted, the following conclusions can be drawn:

Complexity and Diversity of Vulnerabilities: Each type of vulnerability has its own characteristics and requires specific methods of detection and protection.

Need for Continuous Learning: It is essential to constantly improve knowledge and skills in security to effectively counter new threats.

Importance of Early Detection and Prevention: Proactive measures, including code testing and auditing, are necessary and significantly reduce risks.

Future research will primarily focus on an in-depth study of existing vulnerabilities. Following this, it is expected that several insufficiently studied vulnerabilities will be identified, and work will continue in their investigation. As secondary branches, small studies in the field of formal analysis for the automation of SC analytics are possible. Formal verification is a method of mathematically checking the correctness of software.

REFERENCES

1. Solidity. (n.d.). Retrieved May 31, 2024, from https://soliditylang.org/
2. Ethereum. (n.d.). Retrieved May 31, 2024, from https://ethereum.org/
3. Layer3. (n.d.). Retrieved May 31, 2024, from https://layer3.xyz/
4. MetaMask. (n.d.). Retrieved May 31, 2024, from https://metamask.io/
5. Binaryx. (n.d.). Retrieved May 31, 2024, from https://binaryx.com/
6. Ethereum Improvement Proposals (EIPs). (n.d.). Retrieved May 31, 2024, from https://eips.ethereum.org/
7. Yang, Z., & Lei, H. (2020). Lolisa: Formal Syntax and Semantics for a Subset of the Solidity Programming Language in Mathematical Tool Coq. *Mathematical Problems in Engineering, 2020*, Article ID 6191537, 15 pages. https://doi.org/10.1155/2020/6191537
8. Staderini, M., Pataricza, A., & Bondavalli, A. (2022). Security Evaluation and Improvement of Solidity Smart Contracts. SSRN. https://ssrn.com/abstract=4038087 or http://dx.doi.org/10.2139/ssrn.4038087
9. Common Weakness Enumeration (CWE). (n.d.). Retrieved May 31, 2024, from https://cwe.mitre.org/
10. Zaazaa, O., & El Bakkali, H. (2023). A systematic literature review of undiscovered vulnerabilities and tools in smart contract technology. *Journal of Intelligent Systems, 32*(1), 20230038. https://doi.org/10.1515/jisys-2023-0038
11. Soud, M., Qasse, I., Liebel, G., & Hamdaqa, M. (2022). AutoMESC: Automatic Framework for Mining and Classifying Ethereum Smart Contract Vulnerabilities and Their Fixes. *arXiv*. https://doi.org/10.48550/arXiv.2212.10660
12. Common Vulnerabilities and Exposures (CVE). (n.d.). Retrieved May 31, 2024, from https://cve.mitre.org/
13. National Vulnerability Database (NVD). (n.d.). Retrieved May 31, 2024, from https://nvd.nist.gov/

14. Perez, D., & Livshits, B. (2021). Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited. *USENIX Security Symposium*.
15. SWC Registry. (n.d.). SWC-107: Reentrancy. Retrieved May 31, 2024, from https://swcregistry.io/docs/SWC-107/
16. Kadenzipfel. (n.d.). Reentrancy vulnerabilities in smart contracts. GitHub. Retrieved May 31, 2024, from https://github.com/kadenzipfel/smart-contract-vulnerabilities/blob/bedc4dcbbad4310e1e1404636597097359ca7832/vulnerabilities/reentrancy.md
17. Mehar, M., Shier, C., Giambattista, A., Gong, E., Fletcher, G., Sanayhie, R., Kim, H. M., & Laskowski, M. (2017). Understanding a Revolutionary and Flawed Grand Experiment in Blockchain: The DAO Attack. *Journal of Cases on Information Technology, 21*(1), 19–32. Available at SSRN: https://ssrn.com/abstract=3014782 or http://dx.doi.org/10.2139/ssrn.3014782
18. Pcaversaccio. (n.d.). Reentrancy attacks. GitHub. Retrieved May 31, 2024, from https://github.com/pcaversaccio/reentrancy-attacks
19. Ethereum. (n.d.). Ethereum Virtual Machine (EVM) overview. Retrieved May 31, 2024, from https://ethereum.org/en/developers/docs/evm/
20. Lai, E., & Luo, W. (2020). Static analysis of integer overflow of smart contracts in Ethereum. In *Proceedings of the 2020 4th International Conference on Cryptography, Security and Privacy (ICCSP 2020)* (pp. 110–115). Association for Computing Machinery. https://doi.org/10.1145/3377644.3377650
21. OpenZeppelin. (n.d.). Math. Retrieved May 31, 2024, from https://docs.openzeppelin.com/contracts/5.x/api/utils#math
22. Tang, X., Du, Y., Lai, A., & others. (2023). Deep learning-based solution for smart contract vulnerabilities detection. *Scientific Reports, 13*, 20106. https://doi.org/10.1038/s41598-023-47219-0
23. Fang, Y., Wu, D., Yi, X., Wang, S., Chen, Y., Chen, M., Liu, Y., & Jiang, L. (2023). Beyond "protected" and "private": An empirical security analysis of custom function modifiers in smart contracts. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023)* (pp. 1157–1168). Association for Computing Machinery. https://doi.org/10.1145/3597926.3598125
24. CertiK. (n.d.). MEV bot 0xbad incident analysis. Retrieved May 31, 2024, from https://www.certik.com/blog/mev-bot-oxbad-incident-analysis
25. rekt.news. (n.d.). RIP MEV bot. Retrieved May 31, 2024, from https://rekt.news/ripmevbot/
26. Wang, C., Zhang, Y., Li, M., Liu, H., & Zhao, J. (2023). Research on smart contract vulnerability detection method based on domain features of Solidity contracts and attention mechanism. *Journal of Intelligent & Fuzzy Systems, 45*(1), 1513–1525. https://doi.org/10.3233/JIFS-224489
27. Xu, Y., Hu, G., You, L., & Cao, C. (2021). A novel machine learning-based analysis model for smart contract vulnerability. *Security and Communication Networks, 2021*, Article ID 5798033, 12 pages. https://doi.org/10.1155/2021/5798033
28. Akca, S., Rajan, A., & Peng, C. (2019). SolAnalyser: A framework for analysing and testing smart contracts. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)* (pp. 482-489). IEEE. https://doi.org/10.1109/APSEC48747.2019.00071
29. SWC Registry. (n.d.). SWC-116: Block values as a proxy for time. Retrieved May 31, 2024, from https://swcregistry.io/docs/SWC-116
30. Common Weakness Enumeration (CWE). (n.d.). CWE-829: Inclusion of functionality from untrusted control sphere. Retrieved May 31, 2024, from https://cwe.mitre.org/data/definitions/829.html

31. https://consensys.github.io/smart-contract-best-practices/development-recommendations/solidity-specific/timestamp-dependence/
32. ConsenSys. (n.d.). Timestamp dependence. In *Smart contract best practices*. Retrieved May 31, 2024, from https://consensys.github.io/smart-contract-best-practices/development-recommendations/solidity-specific/timestamp-dependence/
33. Solidity by Example. (n.d.). Block timestamp manipulation. Retrieved May 31, 2024, from https://solidity-by-example.org/hacks/block-timestamp-manipulation/
34. Franciscu, S. Y., Ruggahakotuwa, R. K., Samarawickrama, S. W. Y. S., & Lahiru, J. A. D. (2023). GRIFFIN: Enhancing the security of smart contracts. *Trends in Computer Science and Information Technology, 8*(3), 073–081. https://doi.org/10.17352/tcsit.000071
35. Antonopoulos, A. M., & Wood, G. (2018). *Mastering Ethereum: Building Smart Contracts and DApps*. O'Reilly Media. Retrieved May 31, 2024, from https://github.com/ethereumbook/ethereumbook
36. 0xD4v3. (n.d.). Road to Security: Topic 10 - Race Conditions & Front Running. Medium. Retrieved May 31, 2024, from https://medium.com/@0xD4v3/road-to-security-topic-10-race-conditions-front-running-569978979c1b
37. Li, C., Nie, S., Cao, Y., Yu, Y., & Hu, Z. (2020). Trace-based dynamic gas estimation of loops in smart contracts. *IEEE Open Journal of the Computer Society, 1*, 295–306. https://doi.org/10.1109/OJCS.2020.3039991
38. Nassirzadeh, B., Sun, H., Banescu, S., & Ganesh, V. (2023). Gas Gauge: A security analysis tool for smart contract out-of-gas vulnerabilities. In P. Pardalos, I. Kotsireas, Y. Guo, & W. Knottenbelt (Eds.), *Mathematical Research for Blockchain Economy. MARBLE 2022. Lecture Notes in Operations Research*. Springer, Cham. https://doi.org/10.1007/978-3-031-18679-0_9
39. Beylin, M. (2019, January 23). Onward with Ethereum smart contract security. OpenZeppelin Blog. Retrieved May 31, 2024, from https://blog.openzeppelin.com/onward-with-ethereum-smart-contract-security-97a827e47702
40. Vogelsteller, F., & Buterin, V. (2015, November). ERC-20: Token Standard. *Ethereum Improvement Proposals*, (20). Retrieved May 31, 2024, from https://eips.ethereum.org/EIPS/eip-20
41. Shekhar, S. (2018, August 23). Jackpot – A Fomo3D adventure: Examining the data behind the game's first big winner. *TokenAnalyst*. Retrieved May 31, 2024, from https://medium.com/tokenanalyst/jackpot-a-fomo3d-adventure-793f6e94dc75
42. Ballet, G., Buterin, V., & Feist, D. (2023, March). EIP-6780: SELFDESTRUCT only in same transaction. *Ethereum Improvement Proposals*, (6780). Retrieved May 31, 2024, from https://eips.ethereum.org/EIPS/eip-6780
43. Nikolic, I., Kolluri, A., Sergey, I., Saxena, P., & Hobor, A. (2018). Finding the greedy, prodigal, and suicidal contracts at scale (Version 2). *arXiv*. https://doi.org/10.48550/arXiv.1802.06038
44. Akentiev, A. (n.d.). Parity multisig github. GitHub. Retrieved May 31, 2024, from https://github.com/paritytech/parity/issues/6995
45. Etherscan. (n.d.). Address 0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4. Retrieved May 31, 2024, from https://etherscan.io/address/0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4#code
46. Ethereum Contract Security Techniques and Tips. (n.d.). DoS with unexpected revert. Retrieved May 31, 2024, from

https://ethereum-contract-security-techniques-and-tips.readthedocs.io/en/latest/known_attacks/#dos-with-unexpected-revert
47. Ethereum Contract Security Techniques and Tips. (n.d.). Favor pull over push for external calls. Retrieved May 31, 2024, from https://ethereum-contract-security-techniques-and-tips.readthedocs.io/en/latest/recommendations/#favor-pull-over-push-for-external-calls

Володимир Песчаненко, Віктор Рудь
Херсонський державний університет, Херсон, Україна

ОГЛЯД ВРАЗЛИВОСТЕЙ СМАРТКОНТРАКТІВ, НАПИСАНИХ НА SOLIDITY

Смартконтракти, написані мовою програмування Solidity, виконуються на Ethereum Virtual Machine – віддаленій розподіленій системі віртуальних машин для виконання коду. Стан та історія транзакцій смартконтрактів зберігається в блокчейні. Смартконтракти відкривають нові можливості для автоматизації та децентралізації різних сфер: фінансової (DeFi), ігрової (GameFi), юридичної, управлінської та освітньої. Зростання популярності блокчейн-технологій і активне використання смартконтрактів актуалізують питання їхньої безпеки.

Основними цілями цієї статті є огляд наукових досліджень та аналіз важливих вразливостей смартконтрактів. Огляд наукових досліджень дає об'єктивну оцінку поточного стану знань у сфері безпеки смартконтрактів. Аналіз вразливостей необхідний для короткого знайомства з відомими вразливостями й розуміння їхніх наслідків.

Для проведення дослідження використовувалися такі наукові методи: аналіз літератури, класифікація вразливостей, аналіз способів захисту та огляд реальних атак. Розбір історичних випадків атак на смартконтракти створює фундамент для побудови теоретичних моделей атак. Теоретичні моделі атак необхідні для глибокого розуміння природи вразливостей. Після усвідомлення суті атаки формуються рекомендації щодо захисту смартконтрактів: бібліотеки, безпечні шаблони; способи й методи захисту. Крім того, вивчення історичних випадків атак і заходів щодо їх мінімізації або усунення корисно для швидкого реагування на майбутні схожі атаки.

У контексті дедалі складної блокчейн-технології, перевірка смартконтрактів щодо їхніх вразливостей набуває першорядного значення. Під час дослідження зроблено висновки про значущість розвитку сфери безпеки смартконтрактів, систематичного поновлення знань і періодичної повторної перевірки контрактів.

Ця стаття може бути цікава людям, які займаються розробкою смартконтрактів і дослідженнями в галузі кібербезпеки.

Ключові слова: смарт-контракт, Solidity, вразливості, Ethereum, кібербезпека, кібератака, блокчейн, децентралізовані фінанси (DeFi)

Стаття надійшла до редакції 17.12.2023
The article was received 17 December 2023