SemeAd 2025
XXVIII Seminários em Administração

10, 11, 12, 13 e 14 de novembro de 2025
ISSN 2177-3866

# SMART CONTRACTS IN SOLIDITY: a study on good development practices

**MARIANE CARVALHO ESTEVES**
MBA USP ESALQ

**DANIELLE APARECIDA ALCANTARA**
UNIVERSIDADE FEDERAL DE LAVRAS (UFLA)

**JULIANA PAVIANI**
UNIVERSIDADE FEDERAL DE LAVRAS (UFLA)

**JAIRO ANTONIO RESENDE PAVIANI**
UNIVERSIDADE FEDERAL DE LAVRAS (UFLA)

# SMART CONTRACTS IN *SOLIDITY*: a study on good development practices

## 1. INTRODUCTION

Decentralized applications (DApps) represent a significant evolution in the way decentralized applications, or DApps, represent an evolution in the way we interact with technologies. Buterin (2014) describes that by replacing centralized servers with *Blockchain* networks, these applications eliminate intermediaries and distribute the execution of processes among network participants, promoting greater efficiency and autonomy. Complementing this view, Wohrer and Zdun (2018) point out that the execution of smart contracts on the *Blockchain* ensures that transactions take place autonomously and reliably, reinforcing the security and integrity of records.

This change in software architecture, in addition to optimizing the efficiency of applications and transactions, also extends users' control over their own data by eliminating intermediaries and reducing the costs of centralized authorities, as discussed by Zheng et al. (2023). The concept of DApps aligns directly with the principle of decentralization, promoting the distributed management of information without the need for a central authority. This structure provides benefits such as the immutability of records, the elimination of single points of failure and the use of consensus mechanisms to validate transactions, ensuring transparency, auditability and resistance to alterations (Habib et al., 2022).

Among the various platforms aimed at developing these applications, *Ethereum* stands out as the most popular and widely adopted *Blockchain* platform, as pointed out by Connors and Sarkar (2023). In addition, these authors highlight the central role of the *Solidity* language, initially consolidated as the standard for creating smart contracts in this ecosystem. Launched in 2015, *Ethereum* introduced the concept of smart contracts, which are self-executing scripts stored on the *Blockchain*, enabling the creation of DApps that carry out complex transactions autonomously (Buterin, 2014). *Ethereum* has enabled diverse use cases, from decentralized finance (DeFi) to gaming, as highlighted by Zheng (2023).

While decentralization offers many advantages, it can also make it difficult to account for and recover from failures or attacks. For example, smart contracts that contain programming errors can result in irreparable financial losses, as evidenced by notorious incidents in *Ethereum*'s history (Atzei; Bartoletti; Cimoli, 2017). The decentralized nature of DApps has triggered significant security challenges. Code vulnerabilities, inadequate implementation of smart contracts and data manipulation are some of the issues that can compromise the functioning and integrity of applications.

In this context, it is essential to investigate standards and good practices to strengthen security in the development of *Solidity* smart contracts on the *Ethereum* platform. According to Atzei, Bartoletti and Cimoli (2017), adopting security standards, carrying out code audits and applying programming techniques are fundamental measures to mitigate the vulnerabilities associated with DApps. Thus, the purpose of this study is to analyze good practices, and also to contribute to strengthening user confidence in decentralized technology, highlighting the importance of secure approaches.

The main objective of this study is to investigate good practices for the development of smart contracts, with a focus on security. To achieve this objective, the following specific objectives were established: to identify and describe good practices for development in *Solidity*, emphasizing strategies that mitigate risks and vulnerabilities; and to examine case studies of successful smart contracts and those that have suffered vulnerabilities, analyzing the lessons learned and implications for future implementations.

## 2. THEORETICAL BACKGROUND

### 2.1. Fundamentals of decentralized applications

Decentralized applications, or DApps, are systems that use *Blockchain* technology to operate autonomously and distributed, without the need for a single server. This structure ensures greater security and integrity of transactions, as highlighted by Khan *et al. (*2020), by avoiding the need for centralized entities. Another differential of DApps is transparency, as smart contracts are stored publicly on the *Blockchain*, allowing auditing by any network participant (Christidis & Devetsikiotis, 2016), unlike traditional systems, whose operating logic remains opaque to users.

### 2.2. *Ethereum*: platform and components

*Ethereum* is the main platform for developing smart contracts and DApps, and its launch in 2015 was notable for introducing a programming layer on top of the *Blockchain* (Khan et al, 2020). The *Ethereum* Virtual Machine (EVM) supports different languages, attracting a diverse community of developers. In addition to contracts, the tool has popularized fungible and non-fungible tokens, boosting areas such as finance. *Ethereum*'s versatility aims to facilitate innovation and collaboration in a decentralized, secure and immutable ecosystem (*Ethereum* Foundation, 2023). Its native cryptocurrency, Ether (ETH), described by Buterin (2014) as the "fuel" of the network, enables the payment of transaction fees (gas) and encourages validators, ensuring the operation of the ecosystem.

### 2.3. *Blockchain*

*Blockchain*, introduced by Nakamoto (2008) with the launch of Bitcoin, is a distributed data structure that records transactions in an immutable, auditable and decentralized way, without the need for a central authority. Its architecture consists of a continuous chain of blocks that store transaction records validated by consensus mechanisms and protected by asymmetric cryptography, guaranteeing the security and integrity of the data.

The technology also allows the implementation of smart contracts, which automate agreements directly on the network, enabling applications in decentralized finance (DeFi), Internet of Things (IoT). These applications take advantage of the decentralization and immutability of the technology, as highlighted by Zheng *et al. (*2017).

### 2.4. Smart contracts: definition and functionality

Smart contracts are self-executing programs that operate on the *Blockchain*, designed to facilitate, validate and enforce negotiations or executions of a contract, based on previously established conditions. The concept was introduced by Nick Szabo (1994), who described them as digital protocols capable of formalizing negotiations securely and automatically. In the context of *Ethereum*, these contracts are fundamental for automating processes and implementing decentralized applications (DApps). As described by the *Ethereum* Foundation (2023), smart contracts execute pre-programmed rules in the *Ethereum Virtual Machine* (EVM), so that transactions are conducted transparently, securely and immutably, essential characteristics for the reliability of the decentralized ecosystem.

### 2.5. Decentralized development in *Solidity*

*Solidity* is a contract-oriented programming language created specifically for the development of decentralized applications on the *Ethereum* platform. Inspired by languages such as JavaScript and C++, it allows you to write smart contracts that run on the *Ethereum* EVM, logically controlling digital assets and interactions between users. Its syntax and community support have consolidated *Solidity* as the de facto standard in the *Ethereum* ecosystem (*Solidity* , 2024).

### 2.6. Transactions, gas and operating costs

Transactions on the *Ethereum* platform are used to transfer Ether, tokens or interact with contracts. Each operation requires a computational cost measured in units of gas. As described by Buterin (2014), gas protects the network from abuse and encourages code optimization.

The gas fee can vary depending on the complexity of the operation, and poorly designed contracts - with inefficient or unnecessary logic - can become economically unviable for execution (*Ethereum Foundation*, 2023). This scenario reinforces the need to adopt good development practices.

## 2.7. Common vulnerabilities in *Solidity* contracts

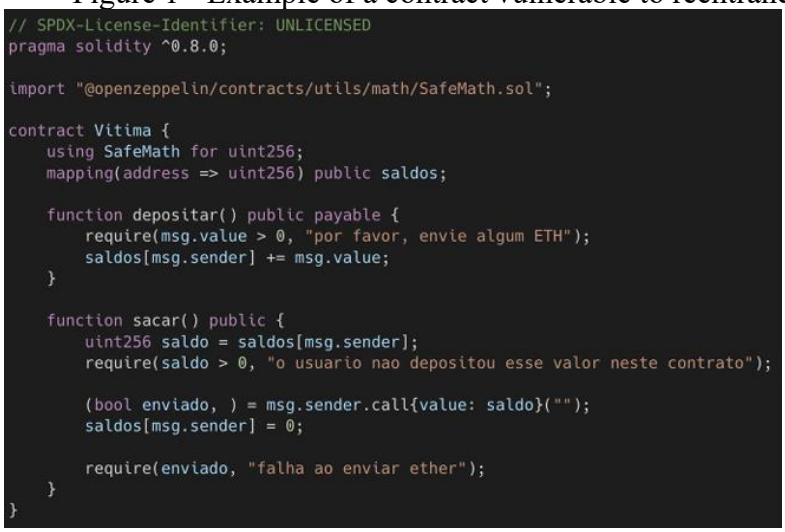
This section addresses two recurring and highly critical vulnerabilities that compromise the security of smart contracts on the *Ethereum* network: reentrancy and access control. These vulnerabilities will be explored considering their causes, impacts and approaches applied for mitigation.

## 2.8. Re-entrance vulnerability

The reentrancy vulnerability occurs when a smart contract allows external calls to be made before updating its internal state, allowing a function to be executed repeatedly in a malicious way. A notorious example of this flaw was the DAO attack, in which the attacker exploited this loophole to make successive withdrawals before the balance was deducted (Soud *et al.*, 2022).

Figure 1 shows a practical example of a vulnerable contract, where the withdraw function performs the transfer with a call before the balance is zeroed, allowing multiple executions of the function.

Figure 1 - Example of a contract vulnerable to reentrancy



```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/utils/math/SafeMath.sol";

contract Vitima {
    using SafeMath for uint256;
    mapping(address => uint256) public saldos;

    function depositar() public payable {
        require(msg.value > 0, "por favor, envie algum ETH");
        saldos[msg.sender] += msg.value;
    }

    function sacar() public {
        uint256 saldo = saldos[msg.sender];
        require(saldo > 0, "o usuario nao depositou esse valor neste contrato");

        (bool enviado, ) = msg.sender.call{value: saldo}("");
        saldos[msg.sender] = 0;

        require(enviado, "falha ao enviar ether");
    }
}
```

Source: Adapted from Demeyer, S.; Rocha, H. Verheijke, D. (2022).

Demeyer *et al. (*2022) point out that this type of attack can be prevented with good practices such as the Checks-Effects-Interactions standard, which reverses the order of operations to ensure security, and the use of mutexes to prevent nested calls.

## 2.9. Access Control Vulnerability

Access control errors arise when critical functions of a contract are accessible to any user, without restrictions based on permissions or identity checks. This flaw can allow malicious users to perform administrative actions, alter sensitive data, perform unauthorized operations and repeat executions, generating operational or financial losses. Macrinici and Gao

(2018) reinforce that security flaws, including access control, are among the most recurrent problems in smart contracts, and stem from both the complexity of *Blockchain* platforms and inadequate development practices.

## 3. METHODOLOGY

This study adopts a qualitative approach, focusing on identifying good practices and analyzing case studies and documentation related to the development of smart contracts on the Ethereum platform in *Solidity*. This section will be organized into two main sections.

### 3.1. Systematic literature review

The systematic review was conducted using the PRISMA (Preferred Reporting Items for Systematic Reviews and Meta-Analyses) protocol, with the aim of ensuring rigor, reproducibility and transparency in the selection of sources (Liberati *et al.* , 2009). The following terms were used as keywords: *Ethereum*, *Solidity* , smart contracts, security, vulnerabilities, DApps, access control, static analysis, *Ethereum* development, *Blockchain*, vulnerability detection and reentrancy, combined with Boolean operators ("AND" / "OR"). The search was carried out on Google Scholar, initially resulting in 47 publications. After applying inclusion and exclusion criteria, such as timeliness, thematic relevance and scientific rigor, 32 studies were selected to make up the theoretical analysis.

### 3.2. Data collection and analysis

The empirical analysis of the study was structured in four complementary stages:

- **Analysis of vulnerability reports**: OpenZeppelin (2024) reports were examined to identify and classify vulnerabilities in smart contracts in terms of type, severity and impact, making it possible to understand the most recurrent risks in the development of DApps.
- **Case study analysis:** Relevant smart contracts were selected from the *Ethereum* ecosystem, with a history of vulnerabilities and market prominence, using platforms such as GitHub, Etherscan, Certik and DappRadar for contextual and technical analysis.
- **Source code analysis:** The codes of the selected contracts were evaluated with the support of the Slither tool, making it possible to detect security flaws and validate the presence (or absence) of good practices in their implementation.
- **Consolidation of practices and tools:** The evidence collected was compared to standards established by the *Ethereum* Foundation and OpenZeppelin, allowing effective practices in the secure development of smart contracts to be identified.

## 4. RESULTS AND DISCUSSION

This section presents the results of the analysis of vulnerabilities found in smart contracts using the *Solidity* programming language on the *Ethereum* platform. To do this, the steps of the methodology described above were followed, which involve analyzing vulnerability audit reports, evaluating the source code of smart contracts and comparing representative case studies.

### 4.1. Analysis and identification of vulnerabilities

The assessment of 100 vulnerabilities, distributed across 15 different contracts, showed that flaws such as reentrancy and access control are still among the most critical. Re-entrance was classified as critical or high in most cases, especially when it involved the movement of

resources. Access control, on the other hand, showed a wide variation in severity, but maintained a high incidence.
According to studies by Khan and Namin (2020) and Praitheeshan *et al. (*2019), these vulnerabilities persist even as development and auditing tools mature. These findings reinforce the need for preventive measures in line with recognized industry standards. Table 1 illustrates the distribution of vulnerabilities by severity and type, highlighting the predominance of logical flaws and the relevance of re-entry and access control attacks.

Table 1 - Distribution of vulnerabilities by severity

| Vulnerability | Total | Critical | High | Medium | Low |
|---|---|---|---|---|---|
| Reentrance | 10 | 1 | 2 | 1 | 5 |
| Access Control | 16 | 4 | 3 | 7 | 7 |
| Integer Overflow and Underflow | 2 | 0 | 0 | 0 | 2 |
| Logical Errors | 42 | 0 | 2 | 10 | 30 |
| Denial of Service (DoS) | 0 | 1 | 5 | 4 | 10 |
| Unverified External Calls | 0 | 1 | 0 | 0 | 1 |

Source: prepared by the authors

It can be seen that reentrancy has a higher incidence in low and high ratings, suggesting that, although mitigated, it is still neglected in certain contexts. On the other hand, access control showed a more balanced distribution between critical, high and medium severities, which indicates failures in the application of security standards. Logical errors, on the other hand, although mostly classified as low severity, represent the largest volume of failures.

### 4.2. Reentry Vulnerability

Based on the vulnerabilities identified in audited and analyzed contracts, it was possible to observe that reentrancy remains one of the main threats to contract security. As highlighted by Khan and Namin (2020), reentrancy represents a critical flaw associated with the malicious execution of recursive functions before updating the contract's internal state. This threat remains recurrent, as reinforced by Praitheeshan *et al.* (2019), and extensively studied by Cai *et al.* (2025), who proposed an approach based on static analysis with fundamental rules in contract patterns, demonstrating that reentrancy is still prevalent even in modern contracts and still suffers frequent attacks even after years of maturity of the *Solidity* language .

### 4.3. Comparative analysis of practices for mitigating Reentrancy

Based on the analysis of OpenZeppelin's audit reports of smart contracts and the in-depth study of widely used decentralized projects such as Uniswap v4, 1inch v2 and Seaport (OpenSea), it was possible to identify and prove a set of good practices that have been fundamental in preventing reentrancy attacks in *Solidity* smart contracts .

In the OpenZeppelin reports analyzed, vulnerabilities related to reentrancy were found at different levels of severity in different smart contracts. The mitigation measures adopted vary from structural corrections to the flow of functions to the direct application of security libraries.

Table 2 summarizes the main findings, categorizing the severity of the vulnerabilities, the proposed solutions and the inclusion or exclusion of the Check-Effects-Interactions standard, one of the most recommended practices for mitigating this type of attack.

Table 2 - Re-entrance vulnerabilities and corrections in audited reports 14.

| Report | Vulnerability | Suggested Solution |
|---|---|---|
| Tesseract Protocol Audit | Reentrancy between functions | Use of nonReentrant |
| Radiant Riz Audit | Reentrancy in emergency function | Updating the pre-emergency status of the transfer |
| Radiant Riz Audit | Lack of reentrancy protection | Use of ReentrancyGuard |
| Avalanche Interchain Token Transfer Audit | Insecure use of transfer and send | Replacement with sendValue |
| Linea V2 Audit | Theft of tokens in the wrong order | Event issued before transfer |
| Liquidity Locker Univ3 Liquidity Locker Audit | Re-entry via sendMessage | Use of call or sendValue |
| Ion Protocol Audit | Logic vulnerable due to incorrect order (CEI) | Logical reordering to follow CEI |
| AVA Teleporter Audit | Transfer before status update | Update of private variables |
| Linea Bridge Audit | Abstract contract modifying state directly | Use of ReentrancyGuard |
| Scroll Batch Token Bridge Audit | Token bridge reentrancy | Add protection for critical functions |

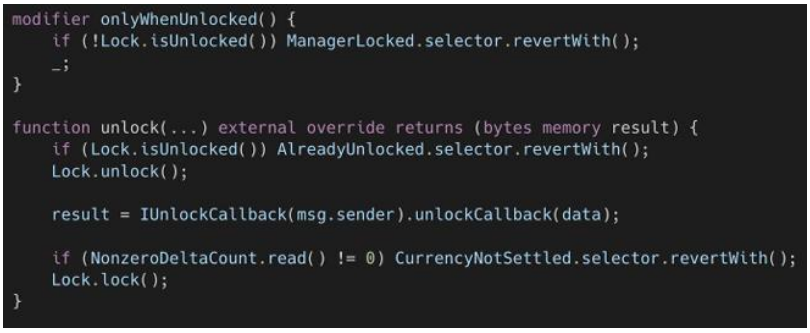
Source: Original research data prepared by the author, 2025.

Despite being consistent with good secure development practices, the solutions adopted in the audited contracts analyzed were, for the most part, applied reactively - in other words, implemented after a vulnerability was identified during the audit process. This highlights not only the importance of audits, but also the lack of robust preventive design in many cases. On the other hand, by analyzing the official repositories of three of the main projects in the *Ethereum* ecosystem - Uniswap v4, 1inch v2 and Seaport - it was possible to identify the application of good practices to mitigate reentrancy proactively and integrated into the architecture of the contracts.

Figure 3 shows that Uniswap implements a manual execution control with a mutex called "Lock", which prevents re-entry into critical functions. The use of the mutex ensures that only one critical execution takes place at a time. This prevents a sensitive function from being called recursively before the previous execution has finished. Thus, even if an external contract tries to induce a reentrance via callback or fallback, the call will be blocked in the onlyWhenUnlocked modifier. This approach is functionally equivalent to nonReentrant, but more adaptable to logic. According to Demeyer *et al. (*2022), the use of "mutex" is one of the

main refactorings that is safe against reentrancy, and is especially effective among multiple public functions.

Figure 3 - PoolManager.sol contract code using customized Mutex.

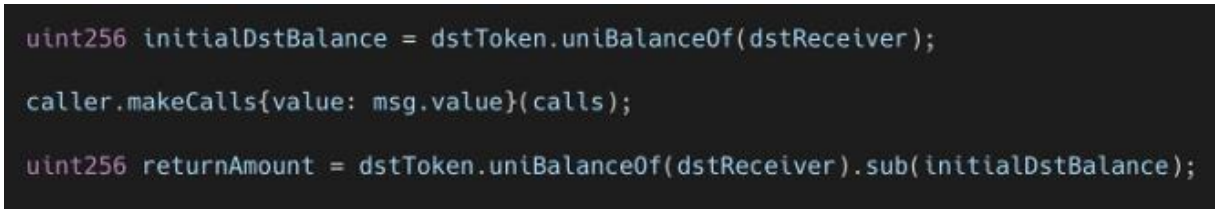


```
modifier onlyWhenUnlocked() {
    if (!Lock.isUnlocked()) ManagerLocked.selector.revertWith();
    _;
}

function unlock(...) external override returns (bytes memory result) {
    if (Lock.isUnlocked()) AlreadyUnlocked.selector.revertWith();
    Lock.unlock();

    result = IUnlockCallback(msg.sender).unlockCallback(data);

    if (NonzeroDeltaCount.read() != 0) CurrencyNotSettled.selector.revertWith();
    Lock.lock();
}
```

Source: Image taken entirely from the source code, available at: https://github.com/Uniswap/v4-core/blob/main/src/PoolManager.sol. Accessed on: 13 Apr. 2025.

Figure 4 shows the correct application of the Check-Effects-Interations pattern. Before the makeCalls external call, the function records the recipient's initial balance and, after execution, calculates and validates the difference based on this value. This approach prevents the logic of the contract from being compromised by indirect reentries, since the state is preserved until the external call is completed. This pattern, highlighted in the study by Demeyer *et al. (*2022), is widely used and recognized for being effective, simple and secure, and is often adopted as a preventative measure in critical contracts subject to reentrancy.

Figure 4 - OneInchExchange.sol contract code using defensive structure with Check-Effects-Interations



```
uint256 initialDstBalance = dstToken.uniBalanceOf(dstReceiver);

caller.makeCalls{value: msg.value}(calls);

uint256 returnAmount = dstToken.uniBalanceOf(dstReceiver).sub(initialDstBalance);
```

Source: Image taken entirely from the source code available at: https://github.com/1inch/1inch-v2-contracts/blob/master/contracts/OneInchExchange.sol. Accessed on: April 13, 2025.

During the analysis of the Seaport protocol, a customized implementation of the "nonReentrant" modifier was identified. Unlike OpenZeppelin's widely adopted solution, which uses a simple Boolean control variable, and the custom "mutex" solution shown in Figure 5, Seaport uses a stored sentinel value to block new executions while a previous call is still in progress. This logic is extended to allow control over the acceptance of native tokens during execution, which demonstrates a specific adaptation to the function. However, the introduction of greater flexibility in this protection requires attention. As reentrancy control now depends on additional parameters and logic inserted into the modifier, it is necessary to assess whether this customization could compromise the security of the function. Therefore, the choice of a customized solution must be carefully evaluated.

Figure 5 - ReferenceReentrancyGuard.sol contract code using a customized modifier.

```
modifier nonReentrant(bool acceptNativeTokens) {
    if (_reentrancyGuard != _NOT_ENTERED_SSTORE) {
        revert NoReentrantCalls();
    }

    if (acceptNativeTokens) {
        _reentrancyGuard = _ENTERED_AND_ACCEPTING_NATIVE_TOKENS_SSTORE;
    } else {
        _reentrancyGuard = _ENTERED_SSTORE;
    }

    _;

    _reentrancyGuard = _NOT_ENTERED_SSTORE;
}
```

Source: Image taken directly from the source code, available at: https://github.com/ProjectOpenSea/seaport/blob/main/reference/lib/ReferenceReentrancyGu ard.sol. Accessed on: April 13, 2025.

Figure 6 shows the practice of validating results immediately after the external call. The swap function is valid if the value returned is in accordance with the minimum expected before completing its execution. This type of verification reduces the risk of unexpected behavior after interactions with third-party contracts by ensuring that the desired effects of the transaction actually occurred before execution was completed. According to Wang *et al. (*2024) the Check-Effect-Interaction (CEI) standard should be respected as a general framework, and that the expected state should be preserved even in scenarios with reentrancy. Validating the expected effects after an external interaction is a way of ensuring that this is happening.

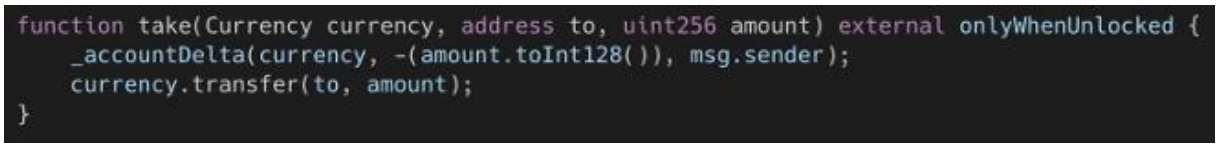
Figure 6 - Post-execution validation on swap function in the OneInchExchange.sol contract.

```
require(returnAmount >= desc.minReturnAmount, "Return amount is not enough");
```

Source: Image taken entirely from the source code available at: https://github.com/1inch/1inch-v2-contracts/blob/master/contracts/OneInchExchange.sol. Accessed on: 13 Apr. 2025

As seen in Figure 7, Uniswap applies the logical separation of business and the movement of funds. The take function exclusively concentrates responsibility for withdrawing amounts, preventing transfers from occurring within broader blocks of contractual logic. This approach favors security, facilitates the verification of behavior and is in line with the principle of Separtions of Concerns (SoC), originally proposed by Dijkstra (1982). This principle, explored by Wohrer and Zdun (2018) as a good development tool in *Solidity* , as it promotes more efficient refactoring, was also shown in the analysis to be a factor that can positively impact the security of the contract and contributes to a more effective analysis of vulnerabilities such as reentrancy.

Figure 7 - Logical separation and transfer of funds in the PoolManager.sol contract.



```
function take(Currency currency, address to, uint256 amount) external onlyWhenUnlocked {
    _accountDelta(currency, -(amount.toInt128()), msg.sender);
    currency.transfer(to, amount);
}
```

Source: Image taken entirely from the source code available at: https://github.com/Uniswap/v4-core/blob/main/src/PoolManager.sol. Accessed on: 13 Apr. 2025

The example in Figure 8 shows the use of the call function as an alternative to the transfer method for sending ETH. This practice is adopted in order to circumvent the gas limitations - a unit that measures the computational cost of operations - imposed by the transfer, which can cause transaction failures, especially in contracts that require more gas to execute. By using calls, developers have greater control over execution and can explicitly define the gas value, promoting greater compatibility with more complex contracts and scenarios (Atzei *et al.* , 2017).

Figure 8 - Using call instead of transfer in the Seaport.sol contract.

```
// Transferência segura de ETH usando call
(bool success, ) = recipient.call{value: amount}("");
require(success, "ETH transfer failed");
```

Source: Image taken entirely from the source code available at: https://github.com/ProjectOpenSea/seaport/blob/main/contracts/Seaport.sol. Accessed on: 13 Apr. 2025

Complementing this qualitative analysis, a scan was also carried out with the Slither tool on the same contracts analyzed - Uniswap v4, 1inch v2 and Seaport - with a focus on detecting possible points of indentation and the results reinforce the qualitative findings: no occurrences of indentation were detected in the Uniswap v4 and Seaport contracts, demonstrating the effectiveness of the practices implemented in these projects. In the case of 1inch v2, the tool indicated three potential occurrences of indentation. Despite this, the alerts were classified with low impact and medium confidence, suggesting that the architecture adopted offers adequate defenses, even though there are points of attention related to external calls. This analysis shows how the combination of manual inspection and automated tools can strengthen vulnerability prevention.

### 4.4. Access control vulnerability

The access control vulnerability was one of the most recurrent in the audited contracts, with 16 occurrences identified during the vulnerability report analysis phase, including critical flaws that make it possible for unauthorized users to manipulate funds or perform administrative functions. This vulnerability persists even in widely audited contracts, such as those for the Uniswap, Taiko Protocol and Compound projects, showing that flaws in this aspect still directly affect the operational security of decentralized applications. According to Khan and Namin (2020), flaws in access control mechanisms represent one of the most prevalent categories of contract vulnerabilities, often associated with the absence of explicit restrictions, with verifications via require or onlyOwner modifiers. Similarly, Jiao *et al. (*2024) highlight inadequate access control as one of the main factors compromising the reliability of smart contracts, especially in sensitive functions that regulate contract administration or the movement of assets.

### 4.5. Comparative analysis of access control mitigation practices

The technical analysis of the audit reports showed the adoption of various practices aimed at mitigating access control flaws in *Solidity* smart contracts . Several vulnerabilities detected were associated with the absence of restrictions on critical functions, allowing unauthorized users to perform sensitive actions that put the integrity of assets and system governance at risk. In response, mechanisms were adopted such as "onlyOwner" visibility modifiers and explicit identity checks using "require (msg.sender == ...)", which represent consolidated practices for restricting access to privileged operations.

In addition to these practices, the investigation identified complementary strategies, such as the null address check "address (0)", used in critical functions, with the aim of avoiding the improper assignment of roles or the execution of invalid calls. Also noteworthy was the validation of the expected sender by means of a whitelist, applied as a way of restricting the execution of sensitive functions to only previously authorized contracts. As highlighted by Khan and Namin (2020), access control failures - such as the absence of explicit ownership and permissions checks - are recurrent and enable attacks that compromise governance and contract funds.

Table 3 presents a summary of the solutions proposed in the reports analyzed, showing a predominance of corrections aimed at applying explicit identity contracts, limiting authorized addresses, and preventing inconsistent states during executions of sensitive functions.

Table 3 - Access control vulnerabilities and corrections extracted from audited reports

| Report | Vulnerability | Suggested solution |
|---|---|---|
| Uniswap v4 Core Audit | Token balance manipulation | Logic adjustment to avoid confusion between native balances and tokens |
| Tesseract Protocol Audit | Function without access restriction | Add a whitelist |
| Taiko Protocol Audit | Falsification of bridge signals | Add checks to validate sender and destination |
| Taiko Protocol Audit | Proposed block vulnerable to token theft | Brings example of author tracking and payment |
| Taiko Protocol Audit | Reuse of signatures | Prevention through the use of unique codes |
| Taiko Protocol Audit | Incomplete resignation of administrative role | Include command to relinquish admin role |
| Euler Vault Kit Audit | Missing input validation | Improve asset input verification |
| Mantle V2 Contracts Audit | Token theft in messaging channels | Blocking addresses that could be exploited |
| Compound III Audit | Unwanted collateral in protocol | Add rules to prevent assets from getting stuck |
| OETH Withdrawal Queue Audit | Essential address not initialized | Check to ensure that the address is not empty |

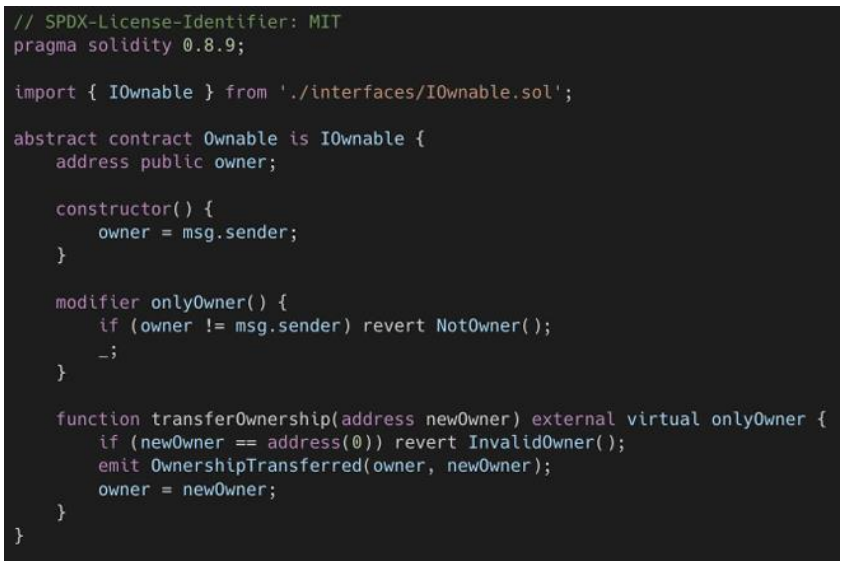
Source: Original research data prepared by the author, 2025.

During the analysis, it was observed that certain vulnerabilities initially classified as reentrancy also involve access control flaws. In specific cases, the absence of restrictions on

who can interact with sensitive functions allows repeated exploitation. This reinforces the interdependence between the two types of flaw and the need for combined mitigation approaches.

Figure 9 shows the access control implemented using the "onlyOwner" modifier, derived from the Ownable contract. This mechanism establishes a single address with authority to perform critical functions, such as administrative changes or transferring funds. By restricting access to these operations, the contract prevents external or malicious agents from interfering with the system's sensitive logic. The use of "onlyOwner" is one of the consolidated practices in the *Ethereum* ecosystem for controlling permissions discussed by Khan and Namin (2020). Additionally, this same approach was also identified in the contracts analyzed for the Uniswap and Axelar contracts, demonstrating their widespread adoption as a protection measure.

Figure 9. Ownable.sol contract code using the onlyOwner modifier for access control.



```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.9;

import { IOwnable } from './interfaces/IOwnable.sol';

abstract contract Ownable is IOwnable {
    address public owner;

    constructor() {
        owner = msg.sender;
    }

    modifier onlyOwner() {
        if (owner != msg.sender) revert NotOwner();
        _;
    }

    function transferOwnership(address newOwner) external virtual onlyOwner {
        if (newOwner == address(0)) revert InvalidOwner();
        emit OwnershipTransferred(owner, newOwner);
        owner = newOwner;
    }
}
```

Source: Image taken entirely from the source code available at: https://*github.com/axelarnetwork/axelar-cgp-Solidity* . Accessed on: 13 Apr. 2025.

During the analysis of the audit reports, it was possible to identify that several access control vulnerabilities could have been avoided by properly validating addresses or by adopting whitelisting mechanisms. These approaches were observed in different ways in the repositories of the Uniswap, 1inch and Axelar projects.

Among the three, Axelar stands out for its implementation of a robust whitelist-based origin validation mechanism, in which only previously authorized contracts or addresses are able to perform critical actions. This control is carried out by directly comparing "msg.sender" with addresses stored in specific internal variables, as in the case shown in Figure 10, using the Software Engineering - 2025 expert variable "KEY_GOVERNANCE". On the other hand, the 1inch and Uniswap projects adopt more traditional access control mechanisms, such as the

exclusive use of onlyOwner. Although functional, these strategies can be limited because they centralize control in a single address, making flexibility difficult in more complex systems, such as environments with multiple participants or interactions between contracts.

Figure 10. AxelarGateway contract code using whitelisting.

```
bytes32 internal constant KEY_GOVERNANCE =
    bytes32(0xabea6fd3db56a6e6d0242111b43ebbb13d11c42709651c032c7894962023a1f909);

modifier onlyGovernance() {
    if (msg.sender != getAddress(KEY_GOVERNANCE))
        revert NotGovernance();
    _;
}
```

Source: Image taken entirely from the source code available at: https://*github.com/axelarnetwork/axelar-cgp-Solidity* . Accessed on: 13 Apr. 2025

Among the vulnerabilities analyzed in the case studies, the possibility of a signature replay attack was identified, in which a valid digital signature can be reused to authorize improper transactions. This flaw compromises the integrity of smart contracts that accept authorizations via signature without restricting them to a single execution. Figure 11 illustrates an example of vulnerable code, in which signature verification is done only with data on the recipient and the amount transferred. Without the use of "nonce", "address(this)" and "chainId", the signature can be reused at another time in the contract or on the network.

Figure 11 - Code vulnerable to the signature replay attack.

```
function permit(
    address owner,
    address spender,
    uint256 value,
    uint8 v,
    bytes32 r,
    bytes32 s
) external {
    bytes32 message = keccak256(abi.encodePacked(owner, spender, value));
    address recovered = ecrecover(message, v, r, s);
    require(recovered == owner, "assinatura inválida");
    _approve(owner, spender, value);
}
```

Source: Image created by the author, 2025.

In contrast, Figure 12 shows a secure implementation. The signed hash includes variables that make the message unique for each context: the "nonce" (unique counter per user), the "address(this)" (contract address) and the "chainId" (network identifier). This approach prevents the signature from being reused in other contracts or networks.

Figure 12 - ERC20Permit.sol contract code implementing a secure solution against replay attacks.

```
bytes32 digest = keccak256(
    abi.encodePacked(
        EIP191_PREFIX_FOR_EIP712_STRUCTURED_DATA,
        DOMAIN_SEPARATOR,
        keccak256(
            abi.encode(
                PERMIT_SIGNATURE_HASH,
                issuer,
                spender,
                value,
                nonces[issuer]++,
                deadline
            )
        )
    )
);
```

Source: Image partially adapted from the source code available at: https://*github.com/axelarnetwork/axelar-cgp-Solidity* . Accessed on: 13 Apr. 2025.

Additionally, another critical vulnerability in smart contracts is the misconfiguration of the role-based access control system. When functions such as assigning or removing permissions are left open or without clear rules, users can maintain privileges they shouldn't have or carry out critical actions without proper authorization. This is especially dangerous in the case of the default administrator role "DEFAULT_ADMIN_ROLE", which has total control over the other roles and needs to be protected.

To avoid this type of flaw, Uniswap has used a secure strategy based on an extension to the OpenZeppelin library called "AccessControlDefaultAdminRoles". According to the OpenZeppelin library documentation (2025), the use of the extension requires a two-step process to transfer the main administrator role, with a mandatory waiting time before the switch takes place. In addition, it prevents the role from being removed directly, ensuring more control and transparency in permissions management.

Finally, the analysis carried out with Slither allowed us to validate the access control mitigation solutions suggested and detailed in this study, reinforcing their adoption in the repositories analyzed. Contracts on the Axelar, 1inch and Uniswap platforms showed consistent use of mechanisms such as AccessControl, role modifiers (onlyOwner, onlyRole) and explicit "msg.sender" checks, and no direct vulnerabilities were detected by the tool at these points. However, despite the good practices implemented, analysis of the Uniswap repository revealed the presence of vulnerabilities classified as access control flaws in specific scenarios, such as the absence of checks on changes of ownership and fee controllers (gas). These flaws, although of low impact, highlight the need for a complementary approach to static analysis, which considers architectural design and the complete authorization flow of contracts.

## 5. CONCLUSION

The analysis carried out in this study allowed us to identify recurring critical vulnerabilities in *Solidity* smart contracts developed for the *Ethereum* network, with an emphasis on re-entry and access control flaws. Based on the literature review, audit reports and code analysis of widely used contracts, it was possible to map patterns of insecurities and highlight effective best practices for their mitigation.

In the case of the reentrancy vulnerability, it was observed that practices such as the Check-Effects-Interactions standard, the clear separation of responsibilities (Separation of Concerns) and the use of the "nonReentrant" modifier - or even its adaptation via "mutex"

type mechanisms - are fundamental to guaranteeing the integrity of critical functions. In addition, continuous validation of shared states between functions complements protection against complex reentrancy attacks, especially when there are multiple interactions between contracts.

The access control failures identified in this research show that the main cause lies in the lack of explicit identity validations, the indiscriminate use of the "onlyOwner" modifier and neglecting to check who makes calls to critical functions. Many of these problems could be avoided with more rigorous "msg.sender" validations, checks against null or unauthorized addresses, and mechanisms based on digital signatures. The comparative analysis between the vulnerabilities reported and the solutions implemented reinforces that the mere presence of a modifier does not ensure security. The adoption of complementary strategies, such as whitelists, the definition of different permission levels for each type of user or role, using tools such as "AccessControl" and logical context checks are essential for robust access control.

In conclusion, most of the vulnerabilities could have been avoided from the initial design of the contracts, with the systematic adoption of good secure development practices. Although Slither has proved useful in evaluating solutions and vulnerabilities in contracts, a more robust approach combines its use with other scanning tools and the performance of specific coding tests, extending the detection of vulnerabilities and strengthening security before deployment on the *Blockchain*.

## BIBLIOGRAPHICAL REFERENCES


1inch. (2021). *1inch v2 Contracts.* Github. Available at: https://github.com/1inch/1inch-v2-contracts. Accessed on: 18 Apr. 2025.

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

Axelar Network (2024). *Axelar CGP Solidity Contracts*. Github. Available at: https://*github.com/axelarnetwork/axelar-cgp-Solidity* . Accessed on: 18 Apr. 2025.

Blockchains. *In the 4th International Conference on Blockchain Economics, Security and Protocols (Tokenomics 2022)*, Open Access Series in Informatics (OASIcs), Volume 110, pp. 3:1-3:15.

Buterin, V. (2014). *Ethereum*: A next-generation smart contract and decentralized application platform. White Paper. Available at: https://Ethereum.org/en/whitepaper/. Accessed on: April 18, 2025.

Cai, J., Chen, J., Zhang, T., Luo, X., Sun, X., & Li, B**.** (2025). Detecting reentrancy vulnerabilities for *Solidity* smart contracts with contract standards-based rules. *IEEE Transactions on Information Forensics and Security, 20*, 3662-3676.

Christidis, K., Devetsikotis, M. (2016). Blockchains and smart contracts for the internet of things. *IEEE Access*, 4: 2292-2303.

Connors, C., & Sarkar, D (2023). Survey of prominent blockchain development platforms. Journal of Network and Computer Applications, 216, 103650. https://doi.org/10.1016/j.jnca.2023.103650

Demeyer, S., Rocha, H., & Verheijke, D. (2022). *Refactoring Solidity smart contracts to protect against reentrancy exploits.* In T. Margaria & B. Steffen (Eds.), ISoLA 2022. Lecture Notes in Computer Science, vol 13702. Springer, Cham.

Dijkstra, E. W., & Dijkstra, E. W. (1982). On the role of scientific thought. Selected writings on computing: a personal perspective, 60-66.

*Ethereum* Foundation (2023). *Ethereum development documentation*. Available at: https://Ethereum.org/en/developers/docs/. Accessed on: April 18, 2025.

Habib, G., Sharma, S., Ibrahim, S. Ahmad, l., Qureshi, S., & Ishfaq, M. (2020). Blockchain technology: Benefits, challenges, applications, and integration of blockchain technology with cloud computing. Future Internet, 14(11), 341. https://doi.org/10.3390/fi14110341

Khan, S., Al-Amin, M., Hossain, H., Noor, N., & Sadik, M. W. (2020). A pragmatical study on *Blockchain* empowered decentralized application development platform. In *Proceedings of the International Conference on Computing Advancements*, pp. 1-9.

Khan, Z. A., & Namin, A. S. (2020, December). *Ethereum* smart contracts: Vulnerabilities and their classifications. In *2020 IEEE International Conference on Big Data (Big Data)* (pp. 1-10). IEEE. https://doi.org/10.1109/BigData50022.2020.9439088

Liberati, A., Altman, D. G., Tetzlaf, J., Mulrow, C., Gøtzsche, P. C., Ioannidis, J. P. A., Clarke, M., Devereaux, P. J., Kleijnen, J., & Moher, D. (2009). The PRISMA statement for reporting systematic reviews and meta-analyses of studies that evaluate healthcare interventions: Explanation and elaboration. BMJ (clinical Research Ed.), 339, b2700. https://doi.org/ 10.1136/bmj.b2700

Macrinici, D., Cartofeanu, C., & Gao, S. (2018). *Smart contract applications within Blockchain technology: A systematic mapping study.* Telematics and Informatics, 35(8), 2337-2354. https://doi.org/10.1016/j.tele.2018.10.004

Nakamoto, Satoshi (2018). Bitcoin: A peer-to-peer electronic cash system. Available at: https://bitcoin.org/bitcoin.pdf. Accessed on April 23, 2025.

OpenSea. (2024). *Seaport Protocol.* Github. Available at: https://github.com/ProjectOpenSea/seaport. Accessed on: 18 Apr. 2025.

OpenZeppelin. (2024). Security audits. *Open Zeppelin Blog.* Available at: https://blog.openzeppelin.com/tag/security-audits/

*Solidity* . (2025). *Solidity documentation (v0.8.29).* Available at: https*://docs.Solidity* lang.org/en/v0.8.29/. Accessed on: 18 Apr. 2025.

Soud, M., Liebel, G., & Hamdaqa, M. (2022). Dataset: A Fly in the Ointment: An Empirical Study on the Characteristics of *Ethereum* Smart Contracts Code Weaknesses and Vulnerabilities *[Dataset]*. *Zenodo.* https://doi.org/10.5281/zenodo.4436232

Szabo, N. (1997). Formalizing and securing relationships on public networks. *First Monday.* 2(9). https://doi.org/10.5210/fm.v2i9.548

Uniswap (2024). *Uniswap v4 Core.* Github. Available at: https://github.com/Uniswap/v4-core. Accessed on: Apr. 18, 2025.

Wöhrer, M., & Zdun, U. (2018). Design patterns for smart contracts in the Ethereum ecosystem. In 2018 IEEE International Conference on Internet of Things (iThings). https://doi.org/10.1109/Cybermatics_2018.2018.00255

Zheng, Zibin, Xie, Shaoan, Dai, Hongning, Chen, Xiangping, & Wang, Huaimin (2017). An overview of *Blockchain*: technology: Architecture, consensus, and future trends. In

2017 IEEE 6th International Congress on Big Data (BigData Congress) (pp. 557-564). IEEE. https://doi.org/10.1109/BigDataCongress.2017.85