DEEP ANALYSIS OF ATTACKS ON SMART CONTRACTS: VULNERABILITIES, METHODS AND COUNTERMEASURES

https://doi.org/10.59982/18294359-23.14-da-15

Gevorg Margarov
NPUA, Institute of ITTE, Head of ISSD Department,
Ph.D. in Technical Sciences, Professor
mgi@polytechnic.am
Narek Naltakyan
NPUA, ITTE Institute, ISSD Chair, MA Student
nareknaltakyan1@gmail.com
Vahagn Gishyan
NPUA, ITTE Institute, Chair of Microelectronic Circuits and Sistems, MA Student
vahagn.gishyan.a@gmail.com
Aghasi Seyranyan
NPUA, ITTE Institute, ISSD Chair, MA Student
aghasi.seyranyan@gmail.com
Aleksandr Martirosyan
NPUA, ITTE Institute, ISSD Chair, MA Student
Alexandr.martirosyan2000@gmail.com

Abstract

Smart contracts are self-executing contracts that have gained significant prominence with the advent of blockchain technology. The unique aspect of these contracts is that the terms of the agreement are directly written into the code, eliminating the need for intermediaries and enabling a high degree of automation. These features have made them particularly attractive in the realm of cryptocurrencies and decentralized applications. However, the inherent complexities associated with the design and implementation of smart contracts have led to a host of vulnerabilities. These vulnerabilities can be exploited by attackers, posing significant security risks. In some cases, these risks have resulted in substantial financial losses, highlighting the need for caution and vigilance. This paper provides a comprehensive overview of the various attack vectors and techniques that attackers can use to exploit these vulnerabilities. It also outlines a range of countermeasures that can be employed to mitigate these risks. The objective of this paper is to serve as a valuable resource for researchers, developers, and users alike. It aims to deepen their understanding of the risks associated with smart contracts and provide guidance on how to mitigate these risks effectively.

Keywords: Smart contract, vulnerabilities, attack techniques, Ethereum, blockchain security, decentralized applications.

Introduction

Smart contracts are programmable, self-executing agreements that operate on a decentralized digital platform, most commonly a blockchain. The concept of smart contracts was first introduced by computer scientist and legal scholar Nick Szabo in 1994. With the advent of Ethereum in 2015, smart contracts found widespread use in creating decentralized applications (DApps) and cryptocurrencies [Rosic]. They are designed to enable trustless, transparent, and secure transactions between parties without the need for intermediaries, such as banks, legal firms, or notaries. Despite their potential to revolutionize various sectors, including

finance, supply chain, real estate, and governance, smart contracts are not immune to security issues. The complex nature of their design and implementation has led to vulnerabilities that can be exploited by malicious actors. These vulnerabilities have resulted in significant financial losses and undermined trust in the technology. As smart contracts continue to grow in popularity, understanding and mitigating these vulnerabilities become increasingly important. This paper aims to provide a comprehensive analysis of the different attack vectors, techniques, and countermeasures relevant to smart contracts. This paper focuses on the vulnerabilities and attacks that are specific to smart contracts, particularly those deployed on the Ethereum platform, which has the largest ecosystem of smart contracts and DApps [Rosic]. However, many of the principles and concepts discussed are applicable to other platforms and implementations. The paper examines various attack techniques, case studies of high-profile incidents, and potential countermeasures to mitigate risks associated with smart contracts.

Smart Contracts

A smart contract is a self-executing contract in which the terms of the agreement between buyer and seller are directly written into code. They are stored on a decentralized digital platform, typically a blockchain, and automatically execute predefined actions when specific conditions are met [Binance]. By removing the need for intermediaries, smart contracts can reduce costs, increase efficiency, and enhance security for various transactions. Smart contracts can be classified into several types, depending on their functionality and the platform on which they are built:

- Financial smart contracts: These contracts facilitate financial transactions, such as token creation, crowdfunding, lending, and derivatives trading. Examples include ERC-20 tokens and decentralized finance (DeFi) platforms.
- Non-financial smart contracts: These contracts are used for non-financial purposes, such as identity management, voting systems, supply chain management, and intellectual property rights management.
- Platform-specific smart contracts: Some smart contracts are designed to function specifically on a certain platform, like Ethereum, Cardano, or Polkadot.
- Platform-agnostic smart contracts: These contracts can be deployed and executed on multiple platforms, providing greater flexibility and interoperability.

Smart Contract Vulnerabilities

Smart contract vulnerabilities arise from flaws in design, implementation, or interaction with the underlying blockchain. The following are some of the most common vulnerabilities:

- Reentrancy attacks: Reentrancy attacks occur when an external contract is called before the state of the original contract is updated. This allows the attacker to repeatedly call the external contract, draining the original contract's funds. The DAO hack in 2016 is a well-known example of a reentrancy attack [Toshendra].
- Arithmetic overflows and underflows: Arithmetic overflows occur when an operation results in a value larger than the maximum value that can be stored in a given data type, causing the value to wrap around. Underflows happen when an operation results in a value smaller than the minimum value, causing the value to wrap around in the opposite direction. These vulnerabilities can be exploited to manipulate token balances or bypass certain conditions in a smart contract.
- Timestamp manipulations: Smart contracts often rely on block timestamps for time-based conditions, such as token vesting or auction end times. However, miners have some influence over block timestamps, which can be manipulated to favor certain outcomes or give the miner an unfair advantage.
- Short address attacks: Short address attacks exploit the fact that some Ethereum clients do not validate the length of input data before processing transactions. Attackers can send transactions with a deliberately shortened address, causing the smart contract to interpret the remaining bytes as part of the transaction data. This can lead to unintended transfers of tokens or other assets.
- Uninitialized storage pointers: Smart contracts use storage pointers to reference data

stored on the blockchain. If a storage pointer is not properly initialized, it can point to unintended locations, allowing an attacker to manipulate or overwrite critical data.

- Delegatecall attacks: The delegatecall function in Ethereum allows a contract to call another contract's function in the context of the calling contract. This can be exploited by an attacker who gains control over the called contract, enabling them to modify the state of the calling contract in malicious ways.
- Front-running attacks: Front-running attacks occur when an attacker observes a pending transaction and submits their own transaction with a higher gas price to ensure it is executed before the original transaction. This can be used to manipulate the outcome of trades, auctions, or other time-sensitive operations in a smart contract.

Attack Techniques and Case Studies

This section examines specific attack techniques and notable incidents that have occurred in the smart contract ecosystem.

- The DAO hack: The DAO (Decentralized Autonomous Organization) was a venture capital fund built on Ethereum in 2016. It suffered a major reentrancy attack, where the attacker exploited a vulnerability in the contract's withdrawal function to drain over 3.6 million Ether (worth around $50 million at the time). The aftermath of the attack led to a controversial hard fork in the Ethereum network, resulting in the creation of Ethereum Classic [Toshendra].
- Parity wallet vulnerability In 2017, a vulnerability in the Parity multisig wallet library contract was exploited by an attacker who managed to steal over 150,000 Ether (worth around $30 million at the time). The vulnerability was related to an uninitialized owner variable in the library contract, which allowed the attacker to take ownership of the contract and subsequently drain the funds [OpenZeppelin].
- Bancor protocol attack In 2018, the Bancor protocol, a decentralized exchange platform, suffered a security breach where an attacker exploited a permission flaw in one of the smart contracts. The attacker was able to call the contract's internal transfer function and bypass the token's standard ERC20 transfer restrictions. This resulted in the theft of over $23 million worth of tokens [Apriorit].
- SpankChain attack In 2018, SpankChain, an adult entertainment platform built on Ethereum, lost around $38,000 in Ether due to a reentrancy attack. The attacker exploited a vulnerability in the platform's payment channel contract, which allowed them to repeatedly call the contract's withdrawal function and drain funds [Roan].
- dForce exploit In 2020, dForce, a DeFi platform, suffered a loss of around $25 million due to an attack exploiting a vulnerability in the platform's lending protocol. The attacker utilized a combination of reentrancy and economic exploits, where they artificially inflated the value of a specific token and then used it as collateral to borrow other assets [Binance].
- bZx flash loan attack In 2020, bZx, a DeFi lending platform, was targeted in a series of attacks that resulted in a loss of over $1 million. The attackers utilized flash loans, a feature that allows users to borrow assets without collateral as long as they are returned within the same transaction. The attackers used these loans to manipulate the price of specific tokens on decentralized exchanges and profit from the price discrepancies in the bZx platform [CoinDesk].

Countermeasures

To mitigate the risk of smart contract attacks, several countermeasures and best practices founded by us can be adopted during the development and deployment phases.

- Delegatecall is a powerful feature of the Ethereum platform that allows contracts to reuse code and modularize their functionality. It works by executing the code of the target contract in the context of the calling contract, which means that the target contract has access to the calling contract's storage, but its own storage is not affected. However, while delegatecall can greatly simplify contract development, it also introduces potential security vulnerabilities. When a contract executes a function using delegatecall, it essentially allows the target contract to execute any code it wants with the calling contract's state. This can lead to unexpected code execution and unintended changes to the calling

contract's state, which can result in security breaches and loss of funds. For example, if contract A has a function that performs a transfer of Ether, and contract B calls that function using delegatecall, contract B's storage will be used for the execution of the code, but the transfer will still be made from contract A's account. This means that anyone who can call the function in contract B can potentially manipulate the transfer and steal funds from contract A. To prevent these types of vulnerabilities, it is important to carefully review and test any contracts that use delegatecall, and to ensure that they are designed with security in mind. This includes implementing proper access controls, auditing any external contracts that are used with delegatecall, and limiting the functions that can be called using delegatecall to only those that are essential for the contract's functionality. By following best practices and being diligent in their code development and testing, developers can use delegatecall safely and effectively to build more efficient and modular contracts.

- Reentrancy is a common software development technique that allows a function to be called repeatedly before the original function execution is finished. While this technique can be useful for certain use cases, it also introduces potential security vulnerabilities, particularly in smart contract development. In a reentrancy attack, an attacker takes advantage of unprotected external calls to repeatedly call a function and drain all of the funds in a contract. This can be a particularly damaging exploit, leading to significant financial losses for the contract owner. To prevent reentrancy attacks, developers can implement a reentrancy guard, which is a modifier that causes execution to fail whenever a reentrancy act is detected. The guard prevents more than one function from being executed at a time by locking the contract, thereby protecting against reentrancy attacks. For example, a simple reentrancy guard can be implemented as follows:

Fig. 1

```
bool locked;

modifier noReentrancy() {
    require(!locked, "Reentrancy detected");
    locked = true;
    _;
    locked = false;
}

function vulnerableFunction() noReentrancy public {
    // function code here
}
```

Reentrancy modifier example in solidity smart contract programming language.

In this example, the noReentrancy modifier sets a locked flag to prevent multiple function executions and reverts execution if a reentrancy act is detected. The vulnerableFunction function uses the noReentrancy modifier to ensure that it can only be called once at a time, protecting against reentrancy attacks. By using a reentrancy guard, developers can help secure their smart contracts against reentrancy

attacks and ensure the integrity of their contract's functionality and funds.

- Solidity is a popular programming language used for smart contract development on the Ethereum blockchain. One of its global variables, tx.origin, returns the address of the original sender of a transaction. However, using tx.origin for authentication purposes can be dangerous and expose a contract to compromise if an authorized account calls into a malicious contract. To prevent tx.origin attacks, developers should avoid using tx.origin for authentication and instead use msg.sender. msg.sender returns the address of the direct caller of the function, which is less susceptible to manipulation by malicious contracts. One key difference between tx.origin and msg.sender is that msg.sender can be a contract, while tx.origin can never be a contract. This means that msg.sender can be used to implement more complex authorization schemes, such as multi-signature wallets or contract-based access controls. For example, consider the following contract that uses msg.sender for authentication:

Fig. 2

```
contract MyContract {
    address public owner;

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

    function doSomething() public {
        require(msg.sender == owner, "Unauthorized");
        // function code here
    }
}
```

Authentication example with msg.sender global variable.

In this contract, the owner variable is set to msg.sender in the constructor, which ensures that only the address that deployed the contract can call the doSomething function. By using msg.sender instead of tx.origin, the contract is more secure and less susceptible to tx.origin attacks. In summary, developers should avoid using tx.origin for authentication purposes and instead use msg.sender. By doing so, they can help ensure the security of their contracts and protect against potential vulnerabilities.

- When choosing a visibility modifier for a function, it's important to consider the intended use and potential security implications. Failure to properly utilize visibility modifiers can lead to unintended state changes and make a contract vulnerable to attacks. Here's a brief explanation of each visibility modifier: Public: A public function can be accessed and called by any account or contract. This includes the main contract, derived contracts, and third-party contracts. Public functions are generally used to expose functionality to other contracts or external users. External: An external function can only be called by an external account or contract. This means that the function cannot be called by the main contract or any derived contracts. External functions are often used for utility functions that don't require access to the contract's state. Internal: An internal function can be called by the main contract and any of its derived contracts. This

means that the function is not accessible to external accounts or contracts. Internal functions are commonly used to implement contract functionality that is not intended for external use. Private: A private function can only be called by the main contract in which it was specified. This means that the function is not accessible to derived contracts or external accounts or contracts. Private functions are typically used for internal implementation details that should not be exposed to external users. It's important to note that functions in Solidity are by default set to public visibility. Developers should carefully consider which visibility modifier is appropriate for each function and explicitly set the visibility to the desired level. This can help prevent unintended state changes and ensure the security of the contract.

- In the Ethereum blockchain, block timestamps are commonly used for various purposes such as generating random numbers, locking funds for a specific time period, and implementing time-dependent conditional statements in smart contracts. However, block timestamps are not always reliable and can be subject to manipulation by validators, making it risky to use them in smart contracts. Validators have the ability to slightly alter timestamps, which can lead to inconsistencies and unpredictable behavior in smart contracts that rely on them. This can result in security vulnerabilities and financial losses for contract users. While it's possible to estimate the time difference between events using block.number and the average block time, relying solely on block timestamps for time-dependent operations in smart contracts is not recommended. Instead, developers should consider using alternative solutions such as block hashes or external time sources. For example, instead of using block timestamps to generate random numbers, developers can use block hashes to ensure a more secure and reliable source of entropy. Similarly, instead of using block timestamps for time-dependent conditional statements, developers can use external time sources such as an oracle or a trusted timestamping service. In summary, while block timestamps can be useful in smart contract development, their use can introduce potential security vulnerabilities. Developers should be aware of the risks associated with block timestamps and consider alternative solutions for time-dependent operations to ensure the security and reliability of their smart contracts.

- In Solidity versions prior to 0.8.0, integers were not wrapped, meaning that they would automatically roll over to a lower or higher number when they reached their maximum or minimum value. This behavior could cause unexpected results in code that relied on integer overflow or underflow. For example, if you decremented 0 by 1 (0 - 1) on an unsigned integer, the result would be the maximum value of that integer type instead of -1 or an error. This is because the integer would wrap around to its maximum value after underflowing. To avoid unexpected behavior and ensure the correctness of their code, developers should be aware of the potential risks associated with integer overflow and underflow and take steps to prevent them. One way to prevent integer overflow and underflow is to use the SafeMath library, which provides safe arithmetic operations for integers. For example, here's how you can use the SafeMath library to subtract 1 from 0:

Fig. 3

```
pragma solidity ^0.8.0;

library SafeMath {
    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        require(b <= a, "Subtraction overflow");
        uint256 c = a - b;
        return c;
    }
}

contract MyContract {
    using SafeMath for uint256;

    function decrement(uint256 x) public returns (uint256) {
        return x.sub(1);
    }
}
```

SafeMath library usage example for solidit version 0.8.0.

In this example, the SafeMath library's sub function is used to subtract 1 from an unsigned integer x. The function checks for underflow and throws an exception if it occurs, preventing unexpected results. By using safe arithmetic operations and avoiding the risks associated with integer overflow and underflow, developers can ensure the correctness and security of their smart contracts.

Conclusion

This paper has provided a comprehensive overview of smart contract vulnerabilities, attack techniques, and countermeasures. We have discussed common vulnerabilities, such as reentrancy attacks, arithmetic overflows, and timestamp manipulation. We also examined notable incidents like the DAO hack, Parity wallet vulnerability, and bZx flash loan attack, illustrating the real-world consequences of these vulnerabilities. The analysis presented in this paper highlights the importance of a security-first mindset for smart contract developers and users. By understanding the various attack vectors and implementing the recommended countermeasures, the risk of smart contract attacks can be significantly reduced. Developers should employ best practices, such as formal verification, static analysis, and fuzz testing, to identify and address vulnerabilities before deployment. Users should be cautious when interacting with smart contracts and consider the security measures implemented by developers. As the adoption of smart contracts continues to grow, ensuring their security becomes increasingly important. Future research should focus on enhancing security analysis tools, developing novel mitigation techniques, and exploring the potential of AI and machine learning for improving smart contract security. Additionally, as the blockchain ecosystem evolves, addressing challenges related to interoperability, privacy, and scalability will be crucial for the successful implementation and widespread adoption of smart contracts.

References

1. A. Rosic, What is Blockchain Technology? A Step-by-Step Guide For Beginners, Jul. 2016, Accessed 24 Feb. 2023, https://blockgeeks.com/guides/what-is-blockchain-technology/.
2. A. Rosic, What is Ethereum?, Oct. 2016, https://blockgeeks.com/guides/ethereum/.
3. Binance, What Are Smart Contracts?, Sep. 16, 2019, Accessed 05 March 2023, https://academy.binance.com/en/articles/what-are-smart-contracts?ref=HDYAHEES.
4. Toshendra Kumar Sharma, Details Of The DAO Hacking In Ethereum In 2016, Oct. 2022, Accessed 08 March 2023, https://www.blockchain-council.org/dao/details-of-the-dao-hacking-in-ethereum-in-2016/.
5. Open Zeppelin, The Parity Wallet Hack Explained, July 19 2017, Accessed 09 March 2023, https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7/.
6. Apriorit, Blockchain Vulnerabilites: Bancor Exchange Hack, Aug. 16, 2018, Accessed 09 March 2023,https://www.apriorit.com/dev-blog/554-bancor-exchange-hack#:~:text=On%20July%209%2C%202018%2C%20the,ether%2C%20to%20a%20personal%20account.
7. Alex Roan, How Spankchain Got Hacked, Mar. 27 2020, Accessed 15 March 2023, https://medium.com/swlh/how-spankchain-got-hacked-af65b933393c.
8. Binance, DForce Confirms the Return of Exploited $3.65m to Their Vaults, Feb. 13 2023, Accessed 15 March 2023, https://www.binance.com/en/feed/post/216587.
9. CoinDesk, The DeFi 'Flash Loan' Attack That Changed Everything, Sep. 13, Accessed 20 March 2023, https://www.coindesk.com/tech/2020/02/27/the-defi-flash-loan-attack-that-changed-everything/

Ներկայացվել է՝
04.04.2023թ.-ին ուղարկվել է
գրախոսման՝ 05.05.2023թ.