Understanding Common Smart Contract Vulnerabilities and the Critical Need for Testing and Audits

Alin-Ciprian ENIȚĂ
National Institute for Research and Development in Informatics - ICI Bucharest
ciprian.enita@ici.ro

Abstract: Following the rapid rise of blockchain technology, smart contracts have emerged as a powerful tool meant to change the way agreements are carried out. Smart contracts can be looked at as self operating agreements that allow predetermined actions to happen when certain conditions are fulfilled, eliminating the need to trust any of the parties involved. In addition, they increase transparency while also increasing the efficiency of operations through reduced need of middlemen. While the potential is huge, along with understanding the benefits this technology brings, one must also acknowledge the responsibility that comes when using it. In order to understand some of the most common weaknesses in smart contracts, one needs to grasp what they are and how they work. Smart contracts are, in fact, pieces of code that run on blockchain networks, automatically performing specific tasks when certain conditions are met. These conditions could be triggered by outside events, interactions from users, or the passage of time. This is where vulnerabilities might occur, potentially causing unintended results. While the fact that smart contracts can't be changed after they're set up helps with honesty and trust, it also presents problems. Finding a code mistake after the contract is deployed and in use might become a very complicated problem to solve. That is why it is paramount to be sure that a smart contract is secure and error free before its deployment. Being aware of the most frequent security dangers can help developers identify and avoid programming approaches that might leave the contracts vulnerable to attacks. In the upcoming sections, the paper will delve into various security hazards, instances of how weaknesses can surface, and tactics or precautionary measures that can be employed to tackle them.
Keywords: blockchain, security, smart contract, hack, audit.

INTRODUCTION

In the realm of smart contracts development, one can constantly and easily take advantage of errors. Thus, addressing security is considered to be of most importance when writing smart contracts. In order to combat the frequent occurrence of certain vulnerabilities, certain best practices and design patterns arised. Similar to other programs, the code that is written is executed precisely, even if sometimes the result does not prove to be what the programmer intended.

https://doi.org/10.54851/v6i1y202407

Furthermore, the public nature of the contracts along with the fact that anyone can interact with them increases the risk of potential exploits. To add even more fuel to the fire, any loss is almost always impossible to recover (Antonopoulos & Wood, 2018).



(Ivanontech, 2021)

The main difference from regular software is that smart contracts run on decentralized networks, which excludes the need for trust. The code is not only fully transparent and open to the public, but it is executed in a predictable way, so that everyone on the network agrees on what happened. Although this is significant and highly desired, it also implies that tiny mistakes can have devastating consequences. Creating smart contracts can prove to be quite a balancing act between new ideas and security best practices like keeping code complexity to a minimum and trying to reuse as much proven code as possible.

REENTRANCY ATTACKS

An important feature of smart contracts is the ability to use code from external contracts. In order to accomplish this, they need to make external calls. Unfortunately, attackers can manipulate these calls and make the victim contract perform malicious code through the use of a fallback function. A popular way of doing this includes making the contracts call themselves again. This type of attack was famously used in the DAO hack incident.

Reentrancy attacks represent for sure a significant category of vulnerabilities. This type of attack happens when a function within a contract is called again before the contract has the chance to change the state according to the results of the previous call. A simplified way to look at it is the attacker finding a way to repeatedly trigger a function in a loop. Obviously, this can cause smart contracts to behave in unintended ways, leading to harmful actions and potentially stealing of funds (Guimarães, 2017).

It must be understood that a reentrancy attack revolves around manipulating the order of actions. An attacker will keep calling a function that implies the transfer of funds, setting the stage for a situation where the internal state of the contract is not fully updated. As a result, attackers can withdraw more funds than they should be allowed to, taking advantage of the mismatch between what is expected and what actually happens.

To prevent reentrancy attacks, certain patterns and techniques like „Checks-Effects-Interactions" should be used. This resumes at making sure a function is performing all internal state changes

and calculations before interacting with external contracts, ensuring that an eventual reentrant call will only be possible after the state is updated (Solidity Lang, 2023).

For a deeper understanding, the present paper will explore the issue addressed step by step. When writing a function, it is advised to always begin with the checks step, which involves validating and checking external conditions or requirements before proceeding with any state changes of interactions. This ensures that the contract is in the expected state and also limits the amount of gas consumed in case the transaction reverts. It is now time for the effects to take place. Following the confirmation of all necessary conditions, the next step should be to perform all the state changes and calculations, which may include updating variables, modifying balances and altering the contract state. At last, only after all the internal state changes are completed, the interactions phase can take place, where one makes the necessary transfers or interacts with the intended contracts. In this way, it becomes certain that, in the eventual attempt of a callback, the internal database has already been accordingly updated.

The presented pattern is crucial for the integrity and security of smart contracts, especially when dealing with external calls. Using this sequence, developers can mitigate the risk of reentrancy attacks and protect against this well-known and frequently utilized type of attack.

INTEGER OVERFLOW AND UNDERFLOW

Like most programming languages, The Ethereum Virtual Machine has specific types of data for integers that come in fixed sizes, which means that an integer variable can hold only a specific set of numbers. For example, uint8 type can only hold numbers from 0 to 255. When trying to store 256 in a uint8, it will actually appear as 0. This means that not properly checking the input from users and allowing calculations that produce numbers beyond the range that a data type can handle will result in severe vulnerabilities (SWC Registry, 2023).

Simply put, integer overflow and underflow vulnerabilities occur when the result of a mathematical operation exceeds the maximum or falls below the minimum value that a data type can hold. The most common reason lies in the incorrect assumptions about the limits of these data types, leading to unintended behavior in the smart contract. These issues might empower a malicious actor to manipulate balances, prematurely access resources, or even disrupt the normal functioning of a contract. Moreover, these vulnerabilities can also affect financial calculations, potentially leading to errors in interest payments, rewards, and other important financial transactions (Antonopoulos & Wood, 2018).

Consider a scenario where a smart contract manages token balance using an integer data type. When exceeding the superior limit which this integer data type can handle, it will result in a totally wrong balance.

To tackle this problem, developers should make sure to double-check the ranges before conducting math operations, hence confirming that the outcome of a calculation falls within the permissible range for the data type. Besides, many blockchain platforms and programming languages provide “safe math” libraries (SWC Registry, 2023), which come with built-in checks, preventing overflow and underflow during arithmetic operations. Incorporating these libraries can significantly reduce the chances of vulnerabilities.

EXTERNAL CALLS: THE GATEWAY TO VULNERABILITIES

An external call is the way a smart contract communicates with other contracts or with sources of information from the outside world like specialized data feeds, allowing them to perform more than just their basic tasks. Reaching out to other functions or data that exist outside, the main contract can highly empower the utility. As stated previously, smart contracts can be considered small computer programs running on a blockchain. They cannot accomplish everything by themselves, especially when they need data

or services that are not available within their own code. Consequently, they need to talk to other smart contracts or external special data providers called "oracles". This talking helps smart contracts work together, become more versatile, and do things that were not initially part of their original plan (Chainlink, 2023b).

Unfortunately, these external calls also bring some tricky security issues, opening doors to potential problems that programmers have to be really cautious about. The introduction of external calls creates extra layers of difficulty and opportunities for hackers. Bad actors can manipulate the way different contracts talk to each other, causing unexpected results or even gain possesion of private information that they should not have accessed.

As a result, attackers might create harmful contracts that communicate with the target contract in an unintended way, exploiting eventual vulnerabilities. Therefore, serious problems can arise from not double-checking or validating the outside data that is being used. Imagine a smart contract trusting data from an external source without checking if that data is trustworthy. That's like blindly following advice from a random stranger without knowing if that person is trustworthy – it could lead to mistakes or manipulation (Antonopoulos & Wood, 2018).

Oracle manipulation is another concerning matter. Oracles are a way to bring external data to a smart contract. However, the danger lies in the quality of the data with which the contract is fed. An oracle can be considered as the connection between digital and the real word. If someone were to mess up that connection, exploitable situations could result. In other words, if a smart contract makes decisions based on incorrect information, the outcome will most likely be undesirable. For example, if a contract makes a trade based on fake price information, the manipulator can make money to the detriment of other users.

Taking all these aspects into consideration, external calls are a powerful feature that should be very carefully approached. To mitigate risks, it is important to follow best practices, write organized code and make sure that the code is tested before deployment. By understanding the potential problems and taking steps to stop the risks before they happen, programmers can make the most of these external conversations, while keeping their smart contracts safe and reliable in the decentralized world.

FRONT-RUNNING ATTACKS: UNFAIR ADVANTAGE

Decentralized finance (DeFi) emerged as a financial technology and brought new dimensions to the smart contract landscape, along with a new set of vulnerabilities. A prime example of this is represented by the front-running attacks. The transparency of the transactions executed on-chain in DeFi applications creates the opportunity for attackers to perform front-running attacks, gaining an unfair advantage. This type of attack refers to the unethical practice of exploiting advanced knowledge of pending transactions, in order to gain advantageous position in a decentralized market (Yazici, 2023). Next, the manner in which front-running occurs will be detailed.

It is important to note that transactions become valid only once a miner successfully completes a consensus mechanism. This gives the miner who solves the block the authority to select which ones of the pending transactions are going to be included in that block. This leads to a classification based on the gasPrice associated with each transaction, meaning whichever has the highest gasPrice is the most likely to be included and executed first. Herein lies the potential vulnerability.

Transaction visibility is the main facilitator factor. In decentralized networks, pending transactions are visible to all participants before they are included in the blockchain, allowing attackers to identify and intercept transactions with profitable opportunities. As a result, attackers monitor the pending transaction pool (mempool) for transactions that can impact market prices (Antonopoulos & Wood, 2018). Once a target is detected, they quickly execute a transaction of their own, sending higher gas fees, in order to make sure they get ahead in the queue.

It is important to understand that miners or validators are incentivized to prioritize transactions with higher gas fees. This entire process allows bad actors to benefit in advance from future market movements.

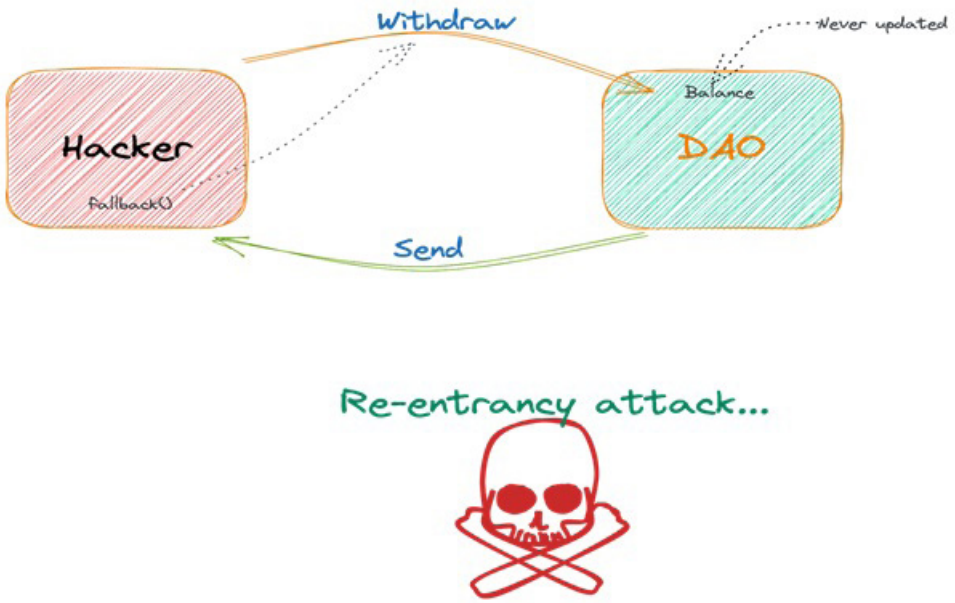
Countering this problem may prove quite difficult from the developer's perspective, but mechanisms such as batched transactions or commit-reveal schemes can be explored. Following a commit-reveal protocol, instead of directly revealing the details of a transaction, users split the transaction process into two steps: commit and reveal.

During the commit phase, the user generates a cryptographic commitment to the transaction details, which includes a unique code that does not reveal the actual content of the transaction. Then, it is broadcasted to the network and included in the blockchain. After a certain predetermined period has passed, the actual details of the transaction are revealed.

These details are also broadcasted, along with the matching cryptographic commitment, allowing the user to prove that the details were indeed the intended transaction. This scheme eliminates the opportunity to anticipate transactions. However, it is important to acknowledge that, while this technique can be very effective against front-running, it may also affect the real time nature of transactions through the delay introduced between the commit and the reveal phases.

Famous smart contract attack

One of the most well-known instances of a reentrancy attack took place in 2016, targeting "The DAO" (Decentralized Autonomous Organization) on the Ethereum blockchain. This event had a significant impact on the blockchain community and sparked crucial discussions about the security and immutability of smart contracts (Antonopoulos & Wood, 2018).



(Ivanontech, 2021)

The victim project was groundbreaking at that time, aiming to establish a decentralized investment fund on the Ethereum blockchain, offering people the opportunity to invest their Ether and, in return, receive DAO tokens. These tokens granted them the power to vote on which projects should receive funding from the organization.

The attack exploited a weakness in The DAO's smart contract code, allowing the attacker to carry out a reentrancy attack. The attacker crafted a malicious contract that invoked The

DAO's splitDAO function. This function enabled DAO token holders to divide their tokens into Ether and a different token known as "reward." The attacker's contract kept calling the splitDAO function in a loop, never giving it a chance to complete. This created a recursive cycle: every time the malicious contract was activated, it would receive Ether from The DAO and instantly call The DAO again, even before the first call was done. This loophole allowed the attacker to drain a considerable amount of Ether from The DAO, by repeatedly taking funds before they could be properly documented (Daian, 2018).

This event led to a substantial theft of the funds invested in the project. It shook the Ethereum community and triggered debates on how to respond. Ultimately, a solution was proposed and carried out through a process called a hard "fork" , which means making a deliberate change to the code of the blockchain in order to reverse the attack's effects and return the stolen Ether to the original investors. However, this decision ignited controversy. It raised questions about the principle of immutability in blockchains and the implications of altering the history of a blockchain. There were a lot of people who disagreed with this approach, which led to the creation of Ethereum Classic, a separate blockchain that remained unchanged from before the hard fork.

The reentrancy attack on The DAO emphasized the vital importance of thorough code audits, rigorous testing, and careful evaluation of potential vulnerabilities in smart contract development. It also showcased the intricate ethical and philosophical dilemmas that arise when addressing weaknesses and making alterations to a decentralized blockchain system.

TESTING SMART CONTRACTS

A process of rigorous testing is required before the deployment of every smart contract, in order to make sure that the code works as expected. Because these contracts often manage high value financial assets, it is important to make sure that even the smallest mistakes are discovered and fixed. This can be done through testing, either automated or manual.

Automated testing implies tools that can automatically analyze the code to identify execution errors. The main advantage of automated testing comes from the utilization of scripts which can be scheduled for repeated execution with minimal human intervention, leading to a less time consuming process compared to manual testing approaches. Thus, this type of tests can prove particularly valuable in scenarios where tests are repetitive and time consuming. Still, it is worth mentioning that automated testing might overlook certain bugs and generate a significant number of false positives. Therefore, automated testing should always be complemented by manual testing. In the case of manual testing, testers will manually compare the actual behavior to its expected behavior, trying to identify potential deviations and bugs. The drawback of manual testing is that it requires significant resources such as skill, effort, time and financial investment. Nevertheless, a human tester can use intuition to unveil edge cases that may not be detected by automated testing tools (Ethereum, 2023a).

A first step would consist in unit testing, which separately evaluates each component, ensuring that it works correctly. This type of testing is useful for checking that functions return expected values and the contract contract storage is updated accordingly. Moreover, every change to a smart contract codebase should be followed by a unit test to ensure that the new logic does not induce errors. Unit testing addresses issues within individual functions, but integration tests assess the entire functionality of different components of the smart contract. The real value of integration testing is shown when a contract follows a modular architecture or interfaces with other deployed contracts during execution. One method of conducting an integration test involves forking a local blockchain, using tools such as *Ganache* or *Hardhat*. This approach allows the user to simulate interactions between his contract and other deployed contracts. Testing a contract on

a local blockchain provides more control over the network and environment itself, allowing the user to adjust gas limits and block times. While testing in a local environment provides valuable insights and is an essential step in the development process, it is also important to conduct a thorough testing on a public testnet, before deploying to the mainnet. Testing on these public testnets allows the user to experience *real world* conditions, such as network latency, varying miner behavior and potential congestion. Consequently, this step should be included as part of the development cycle, in order to enhance the stability and security of the smart contract (Ethereum, 2023b).

However, testing alone can rarely guarantee the absence of bugs in a contract. To further increase the possibility of catching potential vulnerabilities, getting an independent code review such as a smart contract audit is imperative.

SMART CONTRACT AUDITING

It becomes clear that, while the potential benefits of smart contracts are immense, they are far from immune to vulnerabilities and risks. Mistakes in coding, logical flaws, or security vulnerabilities can lead to catastrophic consequences, as shown. That is why smart contract audits play a vital role.

Auditing smart contracts is a meticulous and comprehensive examination of code, logic, and design of a contract. Typically, it is conducted by specialized firms or individuals with expertise in blockchain development and security. These auditors have a deep understanding of common vulnerabilities and attack vectors specific to smart contracts. The primary objective is to identify potential vulnerabilities, security weaknesses, and functional flaws that could compromise the intended behavior of the contract or expose it to exploitation by malicious actors. To accomplish that, skilled professionals thoroughly review the codebase, ensuring it adheres to industry best practices and coding standards. They analyze the logic of the contract to verify that it accurately enforces the agreed-upon terms and conditions. As explained above, one critical aspect of smart contract auditing is the evaluation of the interaction of the contract with external data sources. Auditors assess how the contract processes and validates incoming data, ensuring that it remains resilient against manipulation or inaccuracies in the external information it relies upon. A secondary but very important factor is efficiency. Auditors assess how well the contract utilizes computational resources within the blockchain network, thus aiming to optimize gas consumption and minimize resource wastage.

Throughout the auditing process, experts often engage in various testing and simulation scenarios, subjecting the smart contract to diverse conditions and edge cases in order to uncover hidden flaws or unexpected behaviors that might not have been obvious during the development phase. Documentation and comments within the code are also examined to confirm their accuracy and clarity (Chainlink, 2023a).

Upon completion of the audit, a detailed report is generated. This report presents identified issues, potential risks, and recommendations for improvement. Then developers utilize this feedback to address vulnerabilities and enhance the overall quality of the contract. It is worth noting that the auditing process should not be a one-time event. As a result of the continuous evolution of smart contract and blockchain technologies, the landscape of potential risks changes over time. Consequently, regular audits or reviews are essential to ensure that the smart contract remains secure and effective throughout its lifecycle (Consensys Diligence, 2023).

To summarize, auditing is a crucial practice that enhances the reliability, security, and performance of smart contracts, promoting trust among users, contributing to the broader adoption of blockchain technology, and safeguarding the integrity of agreements and transactions executed through smart contracts. Unfortunately, while audits significantly reduce the risks, there is no any guarantee that they will be completely eliminated.

CONCLUSION

In conclusion, the potential benefits of smart contracts are indeed worth taking into consideration, but their vulnerabilities and risks are equally significant. The role of smart contract audits in identifying coding errors, logical flaws, and security vulnerabilities cannot be understated. Through meticulous examination, audits contribute to the creation of robust and secure smart contracts that fulfill their intended functions without compromising the trust and security that blockchain technology promises. As the blockchain ecosystem continues to advance, safeguarding smart contracts will remain an ongoing imperative, ensuring their effective and secure utilization across various domains.

REFERENCE LIST

Antonopoulos, A. M. & Wood, G. (2018) *Mastering Ethereum: Building Smart Contracts and Dapps*. O'Reilly Media.

Chainlink (2023a) *How To Audit a Smart Contract.* https://blog.chain.link/how-to-audit-smart-contract/.

Chainlink (2023b) *What Is the Blockchain Oracle Problem?* https://chain.link/education-hub/oracle-problem.

Consensys Diligence (19 April 2023) *How to prepare for a Smart Contract Audit?*. https://consensys.io/diligence/blog/2023/04/how-to-prepare-for-a-smart-contract-audit-with-consensys-diligence.

Ethereum (2023a) *Testing Smart Contracts.* https://ethereum.org/en/developers/docs/smart-contracts/testing/.

Ethereum (2023b) *What is Ethereum?.* https://ethereum.org/en/what-is-ethereum/.

Daian, P. (2016) *Analysis of the DAO exploit*. Hacking, Distributed. https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/.

Guimarães, G. (25 May 2017) Reentrancy Attack On Smart Contracts: How To Identify The Exploitable. *Medium*. https://gus-tavo-guim.medium.com/reentrancy-attack-on-smart-contracts-how-to-identify-the-exploitable-and-an-example-of-an-attack-4470a2d8dfe4.

Ivanontech (9 January 2021) DeFi Deep Dive – Smart Contract Security. *Academy Moralis*. https://academy.moralis.io/blog/defi-deep-dive-smart-contract-security.

Smart Contract Weakness Classification (SWC) Registry (2023) *Integer Overflow and Underflow*. https://swcregistry.io/docs/SWC-101/.

Solidity Programming Language (Solidity Lang) (2023) *Security Considerations*. https://docs.soliditylang.org/en/latest/security-considerations.html#use-the-checks-effects-interactions-pattern.

Pratap, Z. (31 August 2022) Reentrancy Attacks and the DAO hack. *Chainlink*. https://blog.chain.link/reentrancy-attacks-and-the-dao-hack/.

Yazici, E. (1 February 2023) Decoding Frontrunning: Understanding the Key Terms and Techniques. *OmniaTech*. https://omniatech.io/pages/decoding-frontrunning-understanding-the-key-terms-and-techniques/.



This is an open access article distributed under the terms and conditions of the Creative Commons Attribution-NonCommercial 4.0 International License.