Finding Vulnerabilities in Smart Contracts

Michele Massetti
big.michelemassetti@gmail.com
Karlsruhe Institute of Technologies
Karlsruhe, Baden-Württemberg, Germany

ABSTRACT

Blockchain is a revolutionary technology that enables users to communicate in a trust-less manner. The most prominent change brought by this technology is the mode of business between organizations: they do not need anymore a trusted third party. It is a distributed ledger technology based on a decentralized peer-to-peer (P2P) network. Since Bitcoin was deployed, many blockchain systems have been born with more capabilities, which have allowed them to fit many different use cases. Smart Contracts, which are programs running on blockchain systems, could extend the potentiality of blockchain from a platform for financial transactions to an all-purpose utility. The development of innovative and prominent applications is a consequence of them, such as NFT marketplaces, music royalty tracking, supply chain and logistics monitoring, voting mechanism, cross-border payments, and many others. Finding bugs and vulnerabilities in them is necessary for assuring their correct behaviour. This paper deals with the way for finding the vulnerabilities in Ethereum blockchain-based smart contracts. We review related works regarding the classification of the most common vulnerabilities and tools which support their detection.

KEYWORDS

Solidity, Software, Vulnerability, Blockchain

1 INTRODUCTION

Nowadays, the major platform for decentralized decentralized finance (DeFi) and applications (dApps) is Ethereum. It can be described as the "internet of Blockchain". Its ecosystem consists of the underlying blockchain, a large variety of smart contracts deployed on it, a wide range of valuable assets.

This growing technology has attaracted many investors, indeed, according CoinGeko, the crypto market's value is standing around $2 trillion. On the other hand, interest in such a market has grown even among malicious attackers. Attacks such as the "Parity Wallet Hack" and the "Decentralized Autonomous Organization Attack" cost millions of dollars simply because of naive bugs in the smart contract code. Blockchain and smart contract technologies have multiple aims, but unfortunately, new applications based on them still contain bugs and multiple vulnerabilities, which cause several issues for the end-users. Most of the use of this technology relates to finance or certifications, therefore integrity, authentication and authorisation in transactions are mandatory.

Colloquium on Vulnerability Discovery 2022, August 1st, 2022, KIT, Karlsruhe, Germany

The research field behind blockchain technology is growing, as well as the one concerning its security and accordingly, many analysis tools were developed. These incorporate various strategies for performing the analyses, concerning the technical aspects of smart contracts, so these would work differently according to the object of the analysis.

Among the many aspects of smart contract, our systematic literature review focuses on studies related to vulnerabilities and analysis tools for their detection. We will try to give an answer to the following research questions:f

- Which are the main vulnerabilities in Smart Contracts?
- Which methodologies are implemented by analysis tools?
- How should we behave for the detection of Vulnerabilities?

In Section **??**, we compare the actual papers and works regarding this topic. Section **??** explains the objective of our analysis: Smart Contracts Vulnerabilities. We give a taxonomy for the main vulnerabilities regarding Solidity. The classification of analysis tools is shown in Section **??**. We discuss about the main strategies implemented by those. In the last Section **??** we adopt the selected tools on the studied Vulnerability, for uderstandig which one is more suitable in wich context.

2 RELETED WORKS

Our work is addressed to define the most common vulnerabilities and methodologies for finding those. The first step was getting in touch with research works which deals with topics as vulenrabilities and analysis tools.

We selected the analaysis tools strating from the reading of surveys and papers as [**?**], [**?**], [**?**], [**?**], [**?**]. Those deal with a great amount of tools, defining their characteristics and the goal is the comparison between those based on selected benchmarks. After the definition, we deepen iur knoledge with their own paper ([**?**] [**?**] [**?**] [**?**] [**?**]). The two most used database for smart cotract Vulnerabilities are [**?**] and [**?**]. Those are run and updated by the community.

3 VULNERABILITIES IN SOLIDITY

An overview of classification of Vulnerabilities in Solidity is given in this section. Classifications by scholars and community taxonomies are the strating point for drowing our own classification. These knoledges are used for having our own classification; we depicted which ones are the most relevant vulnerabilities for our work. The term vulnerability is used in a similar sense than is common in computer security. It refers to a weakness or limitation of a smart contract that may result in security problems. A vulnerability represents even the way the attacker can exploit the contract. This includes locked or stolen resources, breaches of confidentiality or

Table 1: Collection of Vulnerabilities

Vulnerabilities		
Name	DASP10	SWC
Reentrancy	DASP-1	SWC-10
Unexpected Ether balance		SWC-13
Overflows & Underflows	DASP-3	SWC-10
Block Timestamp Manipulation	DASP-8	
Call to the unknown	DASP-10	SWC-11
Transaction Ordering Dependence	DASP-7	SWC-11
Denial Of Service (DOS)	DASP-5	SWC-11
Out of Gas	DASP-5	SWC-12
Unchecked CALL/SEND Return Values	DASP-4	SWC-10

data integrity, and state changes in the environment of smart contracts that were not detected by developers or users, therefor they possibly be exploited by a malicious part.

3.1 Vulnerabilities Classification and examples

We summeries different Vulnerabilities, providing a definition and a real-world example.

Reentrancy. It is one of the vulnerabilities much more exploited during the recent year. Many attacks occured and could steal great amount of many to vulnerable protocol. A reentrancy attack can occur when a function makes an external call to another untrusted contract before it resolves any effects. A recursive call back to the original function is made by the attacker, before the function could updated the state of the state of the variables. The goal is repeating interactions that would have otherwise not run after the effects were resolved. The function withdraw, line 8 to 16 **??**, implements this bug, because the balance of the user is updated after the external call. The attacker, with the usage of a malicious smart contract, can call again the vulnerable function. It is run multiple time, without updating the balance, withrawing all the funds. There are several countermeasures for this type of bug: semaphores logic can be implemented with the modifiers, the update of the variable can be moved before the external call. The external call can be safe if the developer checks the amount of gas used, in order to avoid an attacker can modify the control flow.

```
1  contract EtherStore {
2      mapping(address => uint) public balances;
3      ...
4      //vulnerable function exploited by the attacker
5      function withdraw() public {
6          uint bal = balances[msg.sender];
7          require(bal > 0);
8
9          (bool sent, ) = msg.sender.call{value: bal}("");
10         require(sent, "Failed to send Ether");
11
12         balances[msg.sender] = 0;
13     }
14     ...
15 }
```

Listing 1: Reentrancy

Unexpected Ether balance. An exploites can generated a Dos attack when the smart contract strictly assume a specific token balance. It is always possible to forcibly send ether to a contract (without triggering its fallback function), using selfdestruct, or by mining to the account. Smart Contracts which rely on code execution for every ether sent to the contract can be vulnerable to attacks where ether is sent. In the provided example (**??**), the function play() has a require at line 5, which checks no players can play when the balance reaches the predefined threshold. The function claimReward(), allows the user to withdraw, when the balance reaches the threshold. An attacker can send ether to this contract, for manipulating its balance. If it was greater than the threshold, the functions would always revert because of the require. As a consequence, no one could withdraw the funds and call the function play().

```
1   ...
2   function play() public payable {
3         require(msg.value == 0.5 ether); // each play is
              0.5 ether
4         uint currentBalance = this.balance + msg.value;
5         // ensure no players after the game as finished
6         require(currentBalance <= finalMileStone);
7         // if at a milestone credit the players account
8         if (currentBalance == payoutMileStone1) {
9             redeemableEther[msg.sender] +=
                  mileStone1Reward;
10        }
11        else if (currentBalance == payoutMileStone2) {
12            redeemableEther[msg.sender] +=
                  mileStone2Reward;
13        }
14        else if (currentBalance == finalMileStone ) {
15            redeemableEther[msg.sender] += finalReward;
16        }
17        return;
18  }
19  function claimReward() public {
20    // ensure the game is complete
21    require(this.balance == finalMileStone);
22    // ensure there is a reward to give
23    require(redeemableEther[msg.sender] > 0);
24    redeemableEther[msg.sender] = 0;
25    msg.sender.transfer(redeemableEther[msg.sender]);
26  }
27  ...
```

Listing 2: Unexpected Ether

Overflows & Underflows. The Ethereum Virtual Machine (EVM) defineds fixed-size data types for integers. An example is the identifier uint8, unsign integer over 8 bit, for variables means the variable can only store numbers in the range [0,255]. The attacker can craft the input value of a vulnerable contract's function in order to force a specific operation. In the contract TimeLock in **??**, an attacker could forge a parameter of the function function increaseLockTime, in oder to set lockTime to 0 because of Overflows. So it can call withraw and passing the require at line 17. The most common way for solving these issues is to build or use mathematical libraries which replace the standard math operators. OppenZepplin is a company, dealing with Blockchain technologies. It has done a great job in building and auditing secure libraries which can be leveraged by the Ethereum community. SafeMath library can be found in most of the deployed smart contracts.

```
1  contract TimeLock {
2
3      mapping(address => uint) public balances;
4      mapping(address => uint) public lockTime;
5
6      function deposit() public payable {
7          balances[msg.sender] += msg.value;
8          lockTime[msg.sender] = now + 1 weeks;
9      }
10
11     function increaseLockTime(uint _secondsToIncrease)
           public {
12         lockTime[msg.sender] += _secondsToIncrease;
13     }
14
15     function withdraw() public {
16         require(balances[msg.sender] > 0);
17         require(now > lockTime[msg.sender]);
18         msg.sender.transfer(balances[msg.sender]);
19         balances[msg.sender] = 0;
20     }
21 }
```

Listing 3: Overflows & Underflows

Block Timestamp Manipulation. ispart of the cathegory of vulnerabilities that can be exploited by corrupt miners. A miner cac forge the timestamp by a few seconds, for obtaing benefits. The timestamp dependence vulnerability occurs from a flawed comprehension of timekeeping. It enables the Ethereum network to be detached from the synchronized global clock. The block timestamp can be used as entropy for random numbers, locking funds for periods of time and various state-changing conditional statements that are time-dependent. Since the smart contract permits miners to put up a timestamp within 30 seconds of block validation, this gives a miner more opportunity for exploitation. **??** shows a function (a fallback) which implements the logic of a lottery. The users can send ether to this contract, which accepts just one transaction per block. If the block is a multiple of 15, the user receives all the funds. This function is based on the timestamp of the balck, a malicious miner could adjust the timestamp for getting the win.

```
1    ...
2    // fallback function used to make a bet
3    function () public payable {
4          require(msg.value == 10 ether); // must send 10
               ether to play
5          require(now != pastBlockTime); // only 1
               transaction per block
6          pastBlockTime = now;
7          if(now % 15 == 0) { // winner
8              msg.sender.transfer(this.balance);
9          }
10   }
11   ...
```

Listing 4: Block Timestamp

Call to the unknown. The smart contracts workflow might be manipulated by the invocation of third party or untrusted code. The callee can implemnt malicous code which can be run and modify the normal execution of the calleer. For an untrusted, malicious party, it might be possible to inject code in a way, that can influence the contracts behavior unexpectedly and maliciously. A common case is the reentrancy, which can involve the Solidity functions .call(), .send() and .transfer(). Mistyped direct function calls or using the wrong function signature are classified as this type of vulnerability. The contract CallerContract in **??** is an example of this vulnerability. An attacker could call the function caller, passing as parameter the address of a malicious contract, which implement a malicious fallback. In that case the fallback would be triggered, bacause the contract does not have any function with that name.

```
1  cotract CalleeCotract{
2    fucntion toBeCalled(){
3      //do some operations
4    }
5  }
6
7  contract CallerContract{
8    functionn caller(address a){
9      toBeCalled(a).toBeCalled();
10   }
11 }
12
13 contract Malicious{
14   fallback(){
15     //the attack
16   }
17 }
```

Listing 5: Call to the unknown

Transaction Ordering Dependence. Miners are responsable for the creation and certification of the blocks on the blockchain. Inside those, the transactions are stores, but their order is not guaranteed by design and the miners can easily manipulate that. Consequently, this makes the smart contracts vulnerable to the transaction order dependency, it can be classified as a type of race condition. The vulenrability relies on smart contracs critical decision are based on the order of transaction. Moreover, if two dependent transactions are invoked by the same contract then ordering of transactions must affect the states of blockchain. So we can have a case of race condition, even with a miner without malicious aims.

The smart contract MarketPlace(**??**) implements the logic of a market place. The users can buy an arbitrary amount of stock and the owner can set the price. In this case the variable price is influenced by two functions. If the owner of the contract would be a miner, it could manipulate the order of transaction. The miner-seller can influence the transaction orders and update the price

after the buy request was sent, but before it is processed, effectively increasing the anticipated price for the buyer. The seller can just modify the price, while a user is buying and it can effect its action.

```
contract MarketPlace {
  uint256 price;
  uint256 stock;
  address owner;
  constructor() {
    owner= msg.sender;

  }
  modifier onlyOwner(){
    require(msg.sender==owner);
    _;
  }
  function updatePrice(uint256) onlyOwner {

  }
  function buy(uint256 amount){
    require(stock>=amount);
    require (msg.value >=amount*price);
    stock-=quant;
  }
}
```

Listing 6: Transaction Ordering Dependence

Denial Of Service (DOS). Denial of service attack is a cathegory of attack, which involeves all those one which have the aim to disturb the functionality or stop the execution of the target system. In the context of smart contracts, we can provide some example of this cathegory, which involve multiple strategies for breaking the normal flow execution of the program.

?? shows a simpliefied contract for auction. The function refundUsers() has the responsibility to refund all the bidder. The function contains a loop, so it can get the amount of the bid of the user and refund it. The bug relies on the the require inside the loop. If the transaction returns false, the require fails, so all the transactions are reverted, consequently all the previous refunds are reverted. A malicious contract can be implemented for taking part to the function, but that one reverts every time it receives ether. In this way, the function refundUsers() would always revert and no users can be refunded.

```
contract Auction {
  mapping (address=>uint256) allBids;
  mapping (uint256=>address) users;
  uint256 nUsers;
  uint256 lastrefund;
  ...
  function refundUsers() {

    for(uint256 i = lastrefund; i<nUsers;i++){
      address user=users[i];
      uint256 amount= allBids[user];
      (bool sent, )= user.call{value: amount}("");
      require(sent);
    }

  }
  ...
}
```

Listing 7: Denial of Service/Out of Gas

Out of Gas. In Ethereum, we have to pay for each computational operation, measured in the unit called gas. In the Ethereum documentation, gas is defined as follows: "Gas refers to the unit that measures the amount of computational effort required to execute specific operations on the Ethereum network. Since each Ethereum transaction requires computational resources to execute, each transaction requires a fee. Gas refers to the fee required to conduct a transaction on Ethereum successfully."

The sender pays the transaction fee, previously specifying the gas limit, If the payed amount exceeds the fee, it will receive a refund when the transaction finishes. If the actual execution cost of the transaction exceeds the gas limit, the transaction will fail with an out of gas error. We used **??** for showing an example of DOS, but it implements an out of gas vulnerability too. The amount of gas depends on the number of the for loop's iteration. If the number of iteration grows too much, the function would always revert. A malicious attacker can manipulate the number of iterations, using a great number of wallet for the bid.

Unchecked CALL/SEND Return Values. There a number of ways of performing external calls in solidity. The transfer() method is the most common way for exchanging ether between smart contracts. It reverts if the something goes wrong and function does not succeed. However, the send() and call() function can also be used and, for more versatile external call. These functions return a boolean indicating if the call succeeded or failed. A common bug relies on the return value is not checked, rather the developer expects a revert to occur.

The user can call the function withdrawEmergency() at line 4, **??**, for withdrawing all its funds. The function updates the value of the mapping balances, setting the balance of the user to zero. If the function does not succeed, it does not revert. The funds of the user would be stacked insed the contract. Moreover the amount of ether in the contract is miscomputed.

```
contract Bank{
  mapping (address=> uint256) balances;
  ...
  function withdrawEmergency() public {
    require(balances[msg.sender] > 0);
    uint256 amount= balances[msg.sender];
    balances[msg.sender] = 0;
    etherLeft -= _amount;
    msg.sender.send(amount);
  }
  ...
}
```

Listing 8: Unchecked CALL/SEND Return Values

4 METHODS FOR FINDING VULNERABILITIES

This section is addressed to present the most effective and common tools with their strategies in Ethereum smart contract context. We classify the tools in 2 big cathegories: with and without specification. In the firt case, tools need specifications for running. On the other hand, the former group deals with those which can give a result from the solidity file.

Table 2: Tools without Specifications

Tools Without Specifications	
Name	Keywords
Slither	Static Analysis Scanner
Mythril	Symbolic execution
SmartTest	Symbolic execution
Oyente	Symbolic execution

4.1 Tools without Specifications

Slither. Slither is a smart contract code static analysis framework. It has rapid and reliable security detection algorithms for potential bugs. Slither can be used for automatic vulnerability identification, automated optimization detection, code interpretation, and assisted code review, among other things. For the security analysis, a multi-stage approach is started. From the contract source code, the Solidity compiler generates a Solidity Abstract Syntax Tree (AST), which is utilised as an input to Slither. Slither gathers important contract information such as the inheritance graph, Control-flow graph (CFG), and so on during the early stage. The next stage includes converting the complete code to SlithIR. The code analysis task is then completed by computing a list of pre-defined analyses in the next stage.

Mythril. Mythril is a security tool that examines Solidity-based smart contracts. In order to discover code flaws, Mythril, an open-source programme, employs the symbolic execution technique. Executing smart contract bytecode in a custom constructed EVM is used to look for security issues. To complete its security study, Mythril goes through four major stages. When a programme defect is detected, the input transactions are examined to establish the cause. This security strategy aids in determining the root cause of a program's vulnerability as well as mitigating exploitation. Mythril can discover problems in a contract's source code if a developer provides it.

SmartTest. SmartTest is an Ethereum smart contract safety analyzer. For effectively detecting susceptible transaction sequences, it uses a symbolic execution method. The tool to detect transaction sequences, showing the weaknesses of the examined smart contract, is the project's core problem. As a result, bugs are discovered as the source of multiple transaction interaction. SmartTest's goal is to automatically deliver vulnerable transaction sequences that demonstrate the smart contract's flaws. The fundamental idea is to leverage known vulnerable transaction sequences to develop a statistical model that can be used to drive symbolic execution toward more successfully discovering unknown vulnerabilities. Statistical language models direct symbolic execution, allowing it to select transacion sequences that are likely to reveal vulnerabilities. The first step in this strategy is to perform unguided symbolic execution on existing vulnerable contracts, followed by learning a probability distribution over vulnerable transaction sequences. The detection of the following six types of security-critical vulnerabilities are supported by the tool: integer over/underflow, assertion violation, division-by-zero, ERC20 standard violation, Ether-leaking vulnerability (e.g., unauthorized access to transfer), and suicidal vulnerability (e.g., unauthorized access to selfdestruct). In the paper, the authors focus on just those, without considering vulnerabilities that require analysis of the interaction of multiple contracts to demonstrate the flaws (e.g., reentrancy).

Oyente. Oyente is a symbolic execution tool for finding smart contract security flaws. Oyente analyses Ethereum smart contracts for security flaws that could lead to possible attacks. Oyente not only finds dangerous bugs, but it also looks at every possible execution path. In an experiment conducted by Oyente on 19,366 smart contracts, 8,833 of them were found to be susceptible. The symbolic execution approach uses a mathematical formula to reflect the characteristics of an execution path. Oyente compares the new formula to formulae that contain common bugs to see if both formulas are valid at the same time.

4.2 Tools with Specifications

Echidna. Echidna was created by the same people who created Slither. It's a sophisticated fuzzer for EVM that finds problems in Solidity code. To conduct deep analysis for defects, this tool just requires Solidity propositions and provides a simple user interface (UI) to simplify its results. Echidna tries several combinations of inputs until it succeeds in breaking the given property. Echidna has a few characteristics with Manticore, allowing it to function at the EVM level. It can also be merged to continuous integration (CI) in order to detect code issues while development is ongoing.

Manticore. Manticore is a Solidity audit tool that analyzes smart contracts symbolically. Manti-key core's functions include tracing inputs that terminate a program, logging instruction-level implementation, and offering Python API access to its analysis engine. It features a capability called dynamic symbolic execution that examines binaries and Ethereum smart contracts. The Core Engine, Native Execution Modules, and Ethereum Execution Modules are the three main components of Manticore's architecture. Secondary features include the Satisfiability Modulo Theories (SMT-LIB) module, Event System, and API. In our work, we are using manticore verifier, which deals with the proof of properties given by the developer. The properties are enbedded in the code, written as a solidity function.

SolcVerify. SolcVerify is an Ethereum smart contract source-level verification tool. It accepts Solidity smart contracts and uses modular programme analysis to discharge verification conditions. Because it is built on top of the Solidity compiler, it can reason at the contract source code level. Solc-verify can reason about high-level contract properties while reliably modelling low-level language semantics as a result of this. It is a compiler extension for the Solidity language. It receives a collection of Solidity contracts, including

Table 3: Tools With Specifications

Tools With Specifications	
Name	Keywords
Echidna	Fuzzing, security properties and assertion analysis
Manticore	Symbolic Execution, security properties analysis
SolcVerify	Formal Verification

specification annotations, and discharges verification conditions using the Boogie verifier and SMT solvers. It converts annotated contracts to Boogie Intermediate Verification Language (BIVL)

5 CONCLUSION

ACKNOWLEDGEMENTS

A RESEARCH METHODS

A.1 Part One

Table 4: Results

Results of Analysis							
Vulenrablity	Slither	Oyente	SmartTest	Mythril	Echidna	Manticore	SolcVerify
Reentrancy							
Unexpected Ether balance							
Overflows & Underflows							
Block Timestamp Manipulation							
Call to the unknown							
Transaction Ordering Dependence							
Denial Of Service (DOS)							
Out of Gas							
Unchecked CALL/SEND Return Values							