Solidity Smart Contracts: Vulnerabilities Catalog

Darin Verheijke
darin.verheijke@student.uantwerpen.be
University of Antwerp
Antwerp, Belgium

ABSTRACT

TO DO

CCS CONCEPTS

• **Computer systems organization → Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

KEYWORDS

blockchain, smart contracts, solidity, ethereum, reentrancy

ACM Reference Format:
Darin Verheijke. 2018. Solidity Smart Contracts: Vulnerabilities Catalog. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 3 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

A blockchain is an append-only transactional database where the information is structured together in groups, also known as blocks [3][6]. Each block has certain storage capacities and is chained onto the previous filled block, thus forming a blockchain. Another way to define it is a shared, immutable ledger that records transactions which can be used to track different assets. Most notable uses of this blockchain technology are the cryptocurrencies Bitcoin[7] and Ether [9][4] on the Ethereum network. One important difference between these two blockchain platforms is that Ethereum enables the deployment of smart contracts.

A smart contract is a contract which executes automatically when called upon where the terms between the two parties is written in code (on the blockchain). These contracts then run when a function is called and the conditions for that function are met and can be used to automate executions of agreements without any intermediary party [5][2][6]. In their simplest form a contract is just a collection of functions. Interesting to note is that all smart contract transactions are traceable, transparent and also irreversible [3][6].

A common functionality of smart contracts is the possibility to make calls to another contract on the same blockchain platform. This however needs to be done with caution as untrusted contracts can not only introduce errors but also risks as the contract or call may execute malicious code and exploit vulnerabilities. Every call transfers execution control to the called contract.

One of these dangers when calling an external contract is called Reentrancy and is one of the most well known attacks due to the DAO Attack on June 17th 2016 where around 3.6 million Ether was taken which equated to around $50 million dollar at the time [1]. This exploit is cemented in the history of Ethereum as it resulted into Ethereum being forked into Ethereum Classic and the Ethereum we know today. The original version of this attack involved functions that would be called repeatedly before the first function was finished.

Solidity is one of the major programming languages for smart contracts on Ethereum. To avoid these exploits there have been introduced some best practices. More specifically the function call() was to be replaced by the more safe functions transfer() and send(). However, recently there has been a switch back to the call() function with the introduction of EIP 1884 [8]. Other precautions instead must be taken to prevent reentrancy attacks, one recommendation is making use of a checks-effects-interactions pattern.

In this research internship we will investigate how these calling functions are used in practice and if they are still commonly used in the current smart contracts. For this we have collected a database of around 60K open-source verified smart contracts from Etherscan (dating from July 7th 2021 up until today) to perform our research.

In this research, we expect to find patterns on how contracts use call functions and to potentially find contracts that still include functions that are to be avoided for exploits.

2 BACKGROUND

An introduction to some important concepts such as blockchain, smart contracts and re-entrancy attacks is given. The scope of these concepts will be kept too the Ethereum blockchain and one of its programming languages Solidity.

2.1 Blockchain

A blockchain is a decentralized, distributed and immutable ledger that differs from a typical database in the way that it stores information. All recorded transactions, structured in blocks, are linked together using cryptography. Each block will contain a hash of the previous block, a timestamp and transaction data. As each block contains information from the previous block due to the hash, they will form a chain, thus forming a blockchain.

2.2 Consensus mechanisms

A blockchain reaches consensus when at least 51% of the nodes on the network agree on the next state of the network. These consensus mechanisms are designed to prevent 51% attacks on the network. Currently, Ethereum uses the Nakomoto Consensus, also known as a proof-of-work consensus mechanism which defines the work that

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

miners (network operators) have to do add a new valid block to the chain. The 'work' they do is a cryptographic math puzzle where the winner (i.e. the fastest to solve the puzzle) earns a reward and shares the new block with the rest of the network. An issue with POW is the amount of energy used, Ethereum thus plans to move to proof-of-stake which at a high level has the same end-goal as POW. Very briefly, miners will be replaced with validators who can stake their Ether to activate the ability to create new blocks. This consensus mechanism should also have a stronger immunity to centralization as POS should lead to more nodes in the network. For more information on the POS consensus mechanism and Ethereum their change to it we refer to the documentation. [?]

2.3 Ethereum & Smart contracts

Ethereum and Ether, the currency used to complete transactions on the network differs from Bitcoin in that it enables the deployment of smart contracts and decentralized application also known as dApps. Ethereum has two account types, Externally-Owned and contracts. Externally-owned accounts can be controlled by anyone who has the private keys to this account while smart contracts are deployed to the network and are controlled by code. Smart contracts can be used to create a range of dApps. In this paper we will focus on the smart contract itself which in it's most basic form is a piece of code that runs on the blockchain and guarantees the same output for everyone that runs them. An important feature of smart contracts is that when a function is called and the conditions for that function are met there is an automatic execution of the set agreements without any intermediary party.

Let's look at the vending machine example Nick Szabo, who first coined the term 'Smart contract'. A simple vending machine will take in coins and via a simple mechanism dispense change and the output we selected. How a vending machine removes the need for a vender employee, smart contracts remove the intermediary party. Some important properties of smart contracts is that they are immutable and deterministic. Immutable because once deployed due to the nature of how a blockchain works, the code can not change of the smart contract. The only way to modify a smart contract is to deploy a new instance (and thus a new smart contract). Deterministic in the way the outcome after is identical for everyone who, given the same transaction parameters and state of the Ethereum blockchain, executes the contract.

Important to note, contracts are only run if they are called by a transaction. A contract can call another contract that in its turn can then call another contract however the first part of this chain will always be a transaction called by an Externally-Owned account. All transactions will either successfully terminate or revert. While a contract can not be changed, it can be deleted, removing the code and its storage from its address leaving an empty account. Any transactions sent there will not result in any execution of code.

Executing a transfaction requires a fee which is called the gas fee. It refers to the amount of computational effort required to execute specific operations, these fees are paid in Ether. It's part of the reward miners get in exchange for their service.

Anybody with enough Ether can deploy and write a smart contract to the network. Deployment of a smart contract is also considered a transaction and thus also requires a gas fee.

2.4 Solidity

One of those smart contract languages in which contracts are coded is Solidity, which is an object-oriented, high-level language which has syntax comparable to C++. We won't go in the details of this programming language but instead will look at and describe the call, send and transfer functions which will be relevant for understanding our research, all of these three functions are used as a way to exchange Ether.

The call function is a low-level interface for sending a message to a contract and is a way to send Ether via calling the fallback function, the call function allows for some vulnerabilities which we will go more in depth about later. The send function also made it possible to send an amount to a specific address however it had two small issues. Only a small amount of gas is sent along, which allows for only one single event at max and thus it can't be a contract that changes state variables or does another call to another contract. While this does make it safe, it does cause developers to default back to the unsafe call function to implement calls to other contracts. The other issue it had is that an error would return false instead of propagating the error. The transfer function was introduced which fully propagates errors on throw. However, the option to specify the amount of forwarded gas was not implemented in this function which made it very similar to the send function. Let's look at a simple example of two contracts, a send and a receive contract.

```
contract ReceiveEther {
    function () public payable {}
}

contract SendEther {
    ReceiveEther private receiveAdr = new ReceiveEther();

    function sendEther(uint _amount) public payable {
        if (!address(receiveAdr).send(_amount)) {
            // handle fail send
        }
    }

    function callValueEther(uint _amount) public payable
    {
        require(address(receiveAdr).call.value(_amount).
    gas(35000)());
        }
    function transferEther(uint _amount) public payable {
        address=(receiverAdr).transfer(_amount);
    }
}
```

Listing 1: Ether exchange

2.5 Reentrancy attack

As mentioned before, the call function has some vulnerabilities. Every call to another contract transfers execution control to this called contract. Untrusted contracts may introduce errors as the contract or call may execute malicious code or exploit vulnerabilities. One of these major vulnerabilities is a reentrancy attack which takes advantage of the transfer of execution control by making recursive calls back to the original contract and repeating transactions. In a worst case scenario, this exploit drains all Ether from a contract.

The two main types of reentrancy attacks are single function and cross-function.

2.5.1 Single function reentrancy attack. This version repeatedly calls the involved function before the first invocation of the function is finished. We can see how this can be exploited in following function

```
mapping (address => uint) private userBalances;

function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];
    (bool succes, ) = msg.sender.call.value(
     amountToWithdraw)("");
    require(success);
    userBalances[msg.sender]
    }
// Fallback function which gets executed
function () public payable {
    withdrawBalance()
}
```

Listing 2: Single function reentrancy attack

In this simple example an attacker can recursively call the withdraw function and drain the whole contract as the user's balance is only set to 0 at the very end of the function.

2.5.2 Cross-function reentrancy attack. When a function shares a state with another function there is a possibility of a cross-function reentrancy attack. These are harder to detect.

```
mapping (address => uint) private userBalances;

function transfer(address to, uint amount) {
    if (userBalances[msg.sender] >= amount) {
        userBalances[to] += amount;
        userBalances[msg.sender] -= amount;
        }
    }
function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];
    (bool succes, ) = msg.sender.call.value(
     amountToWithdraw)("");
    require(success);
    userBalances[msg.sender] = 0;
}
```

Listing 3: Cross-function reentrancy attack

Here the attacker will call the transfer function when the code is executed on an external call in withdrawBalance, again the user their balance is not yet set to 0 and thus they will be able to transfer tokens again. A simple solution to both these types of attacks is updating the balance before transferring control to another function or contract.

2.6 Examples of Reentrancy

2.6.1 DAO attack.

3 RESEARCH

4 RESULTS

REFERENCES

[1] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *Principles of Security and Trust*, Matteo Maffei and Mark Ryan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 164–186.

[2] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. 2016. Formal Verification of Smart Contracts: Short Paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security* (Vienna, Austria) *(PLAS '16)*. Association for Computing Machinery, New York, NY, USA, 91–96. https://doi.org/10.1145/2993600.2993611

[3] Santiago Bragagnolo, Henrique S C Rocha, Marcus Denker, and Stéphane Ducasse. 2018. SmartInspect: Solidity Smart Contract Inspector. In *IWBOSE 2018 - 1st International Workshop on Blockchain Oriented Software Engineering*. IEEE, Campobasso, Italy. https://doi.org/10.1109/IWBOSE.2018.8327566

[4] Ethereum Foundation. 2014. Ethereum's white paper. (2014). https://ethereum.org/en/whitepaper/

[5] Ari Juels, Ahmed Kosba, and Elaine Shi. 2016. The Ring of Gyges: Investigating the Future of Criminal Smart Contracts. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 283–295. https://doi.org/10.1145/2976749.2978362

[6] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 254–269. https://doi.org/10.1145/2976749.2978309

[7] Satoshi Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. (2008).

[8] Martin Holst Swende. 2019. *EIP-1884: Repricing for trie-size-dependent opcodes*. https://eips.ethereum.org/EIPS/eip-1884

[9] Gavin Wood. 2018. Ethereum: A secure decentralised generalised transaction ledger. (06 2018), 1–39.