**Article Notes**

Name of the Article: Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited

Written by: Daniel Perez and Benjamin Livshits, Imperial College of London

**Ethereum**

What is it?: Creation, differences to Bitcoin, DAPPS, Smart Contracts

Ethereum was conceived by the programmer Vitalik Buterin in 2013, and the network went live on July 30th, 2015. Like the BitCoin Network, The Ethereum Network is also a network of many computers (called nodes) spread out worldwide that run the Ethereum platform. Unlike Bitcoin, which its goal was to decentralize money, Ethereum seeks to give users the capability to decentralize any service one can think of, such as social media like EtherTweet, renting computing power like Golem and more. Specifically, Ethereum's goal is to be a decentralized "World Computer" that can give decentralized services to users.

These decentralized services are called decentralized applications, or DAPPS. Like any application DAPPS have a front-end which can be written in any programming language, and a back-end which is written in the Ethereum programming language known as Solidity (or other languages such as Vyper and more). These small programs that are written in the backend are called "Smart Contracts" which are integrated into the Ethereum block chain. The "Smart Contracts" are just like any private user account: they have a balance of the Ether related to the account, a public key, can call a different smart contract, can send and receive money and more (but don’t have a private key because they are not controlled by anyone). The "Smart Contracts", as mentioned before, are lines of code that have the structure of an actual contract ("if", "where", "then") such that when all terms have been met the "Smart Contract" executes its code, and while doing so updates the block chain of the transaction that has been made.

These Smart Contracts are compiled through the Ethereum Virtual Machine (EVM) and are represented with simple stack operations like push and pop, and blockchain specific operations like ADDRESS, BALANCE and more.

The Currency Used: Ether

Running the computers (electricity) and maintaining the data on the blockchain (memory) costs money. Therefore, the computers that are a part of the Ethereum Network receive as an incentive the Ether currency. Now, anyone who wants to build a software application on the Ethereum Network will have to pay with the Ether currency to use that computing power and the space that is required. The price is determined by a built-in pricing system called GAS, which lists the amount of Ether needed for the computing power that needs to be provided by the network to complete a transaction.

Differences to Bitcoin:

|  |  |  |
| --- | --- | --- |
|  | **Bitcoin** | **Ethereum** |
| Vision | Digital money, decentralized virtual currency | Smart Contracts, decentralized applications |
| Founder | Satoshi Nakamoto | Vitalik Buterin |
| Release Date | 2009 | 2015 |
| Max Amount | 21 million | 18 million per year |
| Hash Function | SHA256 | Keccak-256 |
| Block Creation | 10 min | 15 seconds |

**The Article**

introduction

This article aims to contrast the vulnerabilities reported in smart contracts on the Ethereum blockchain. The data used to analyze these vulnerabilities are the 23,327 smart contracts that have been marked as vulnerable by those 6 other papers. While most studies have focused on detecting vulnerable contracts, in this paper the writers focus on finding how many of their vulnerable contracts have actually been exploited.

The Vulnerabilities

1. **Re-Entrancy (RE):** Re-Entrancy refers to the capability of running multiple invocations of a function concurrently while doing it safely, meaning without any unanticipated behavior. When a contract calls another account, it can choose the amount of gas it allows the called party to use. If the target account is a malicious contract and the gas budget is high enough to run its code, then this contract can try and call back the first contract. If the first contract is not Re-Entrancy safe, for example because it did not update his internal state and the current balances information, then the malicious contract can use this vulnerability to drain funds from that contract. This vulnerability was used in TheDAO.
2. **Unhandled Exceptions** **(UE)**: The Solidity programming language has a low-level operation called "send" which is used to send Ether to another account. This function does not raise an exception if it has failed, but rather returns a Boolean. If this value is not checked, the caller can continue to call the function that holds this vulnerability which can lead to inconsistencies.
3. **Locked Ether (LE)**: Ethereum smart contracts like any other account can hold Ether. However, there can be multiple reasons for the Ether that is owned by a contract to be locked, out of reach. For example, if there are 2 smart contracts, such that contract A depends on contract B for making a transaction, and contract B has used "SELFDESTRUCT", then now contract A cannot send out anymore transactions and its Ether is locked and unreachable. The self-destruct is a function that can be implemented into the contract code, such that when invoking it, will delete the contract code and send the remaining funds that the contract owned to a specific address. It is important to notice that the data of a self-destructed contract and all the transactions that have been done with it are not erased from the block chain.
4. **Transaction Order Dependency (TO):** In Ethereum, multiple transactions are included in a single block, which means that the state of a contract can be updated multiple times in the same block. If the order of two transactions calling the same smart contract changes the final outcome, an attacker could exploit this property. Essentially, this attack is a race condition attack that can happen if there are 2 or more transactions with the same contract such that the order of mining these transactions can result in different outcomes. For example, let there be a contract that sells something at an initial price of 100. We will look at the following scenario: a client wants to buy from the contract, and he knows the price is 100. The owner of the contract decides to switch the price to 150. Now, if the client's transaction is mined first and then the owner, then nothing wrong has happened. But, if the owner's transaction was mined first and only then the client's, then the client bought at a higher price.
5. **Integer Overflow (IO):** Integer overflow and underflow a=is a common type of bug in many programming languages but in the context of Ethereum it can have very severe consequences. For example, if a loop counter were to overflow, creating an infinite loop, the funds of a contract could become completely frozen. This can be exploited by an attacker if he has a way of incrementing the number of iterations of the loop, for example, by registering enough users to trigger an overflow.
6. **Unrestricted Action (UA):** Contracts often perform authorization checks, by checking who is the sender of the message and restricting him to only specific actions that he can do. An example of a crucial authorization check that needs to be done is that only the owner of a contract can order it to self-destruct, or to give the contract a new owner.

Definitions

* **Vulnerable:** A contract is vulnerable if it has been flagged by a static analysis tool as such. We will see later that some contracts that have been flagged vulnerable are false positives.
* **Exploitable:** A contract is exploitable if it is vulnerable, and the vulnerability could be exploited by an attacker. For example, if the vulnerability that was flagged by the static analysis tool is in a function that can only be called by the owner, then that contract is vulnerable, but not exploitable.
* **Exploited:** A contract is exploited if it received a transaction on the Ethereum network which triggered one of its vulnerabilities. Therefore, a contract can be vulnerable or even exploitable without having being exploited.

Dataset

Our dataset is comprised of a total of 821,219 contracts, of which 23,327 contracts have been flagged as vulnerable to at least one of the six vulnerabilities described before.

Chart, histogram

Description automatically generatedout of the 800,000+ contracts in the dataset, how many were examined by 1 static analysis tool, how many were examined by 2 and so forth.

Chart, histogram

Description automatically generated out of the 800,000+ contracts in the dataset, how many were flagged vulnerable by 1 static analysis tool, how many were flagged vulnerable by 2 static analysis tools and so forth

The writers also check the overlap between contracts in the dataset:

1. How many contracts have been analyzed by multiple tools as depicted in the 1st image.
2. How many contracts were marked vulnerable by multiple tools, as depicted in the 2nd image.

The overlap between tools in both graphs is relatively small, and the suggestion for this is because some analysis tools work on EVM code, while others work on Solidity Code. There is also a big contradiction between different tools used, especially between 2 tools that one works with EVM and the other with Solidity, which may suggest at the difficulty of building a static analysis tool that works with EVM.

Methodology

To check for the different vulnerabilities, the writers perform a bytecode-level transaction analysis, where they look at the code executed by the contract when carrying out a particular transaction. They extract the execution traces for the transactions potentially affecting contracts of interest and use the EVM's debugger to replay transactions while tracing executed instructions. To analyze the traces, they encode them into a Datalog representation. V is a set of the program variables and A is a set of Ethereum addresses.

Datalog is a language implementing first-order logic with recursion (first order logic has relations, variables, quantifiers etc)

The Datalog methodology created by the writers is as follows:

1. Datalog facts: EVM bytecode instructions are interpreted into facts, for example an EVM instruction like CALL which calls an address *a* with *p* Ether is written as call (a1, a2, p) where a1 is the address of the caller, a2 the address of the callee and p the amount of Ether that is transferred.
2. Datalog rules: a set of rules that if certain facts have happened, then the rule is true. For example, if it is a fact that call(a1, a2, p) , then call\_flow(a1,a2,p) is true.
3. Datalog queries: A list of 6 queries, one for each vulnerability that this paper is checking. For example, in order for the Re-Entrancy query to be true, call\_flow(a1, a2, p1) and call\_flow(a2, a1, p2) must be true such that a1 != a2.

The Analysis

1. **Re-Entrancy:** We look for CALL instructions and record the fact call(a1,a2,p). We also look for CREATE instructions, (a contract can also create a new contract with this instruction and execute its code, which can also be used for this attack) and treat it the same way as CALL. We notice that this analysis can have a problem. If we mark a Re-Entrancy vulnerability it means that there was a valid trace of calls leading to our Re-Entrancy query being true, but it does not mean that the contract is in fact vulnerable to Re-Entrancy because we don’t know if the call back is malicious or not. This means that with this method we can get false positives in the analysis. This can be clearly seen with following example: call(a1,a2,p1), call(a2, a3, p2), call(a3, a1, p3). The rule call\_flow(a1,a3,p) is there for true which leads to the Re-Entrancy query to be true, but we cannot tell for certain if a1 is vulnerable to this attack.
2. **Unhandled Exceptions:** We look for CALL instructions and record as facts call\_result(v, n) where v is the result of the call and has a value of n. We also look for JUMPI instructions which is an instruction to perform conditional jumps. We then record as facts in\_condition(v) if v was used as a condition in JUMPI. There for, we can look for all call results variables that were never used to influence a condition. We notice that this analysis can have a problem. We may get false negatives if the call result is used in a condition, but the condition is not enough to prevent an exploit of this vulnerability.
3. **Locked Ether:** As there are many reasons that can cause contracts to lock their Ether, the writers decided to focus on the case where a contract relies on an external contract which does not exist anymore, meaning it used SELFDESTRUCT. The reason for this is because this case has the largest financial impact on Ethereum. To analyze this vulnerability, we look for the instruction DELEGATECALL which is used when one contract calls another contract. The next important thing to notice is the behavior of the EVM when it tries to call a contract that does not exist anymore. What happens is that instead of marking the call is a failure, it is considered a success and the EVM proceeds to the next instruction. There for we collect Datalog facts about all the values of the program counter before and after every DELEGATECALL instruction. Firstly we record the program counter value at which the call was executed with call\_entry(i1,a), *i1* is the value of the PC and *a* the contract being called. Then we record the PC value after exiting, meaning what the PC value was at the callee return and mark this as call\_exit(i2). If the called contract does not exist anymore, then i1 + 1 = i2 must hold. The correctness of this analysis for this **specific** vulnerability is valid.
4. **Transaction Order Dependency:** To check for exploitation of this vulnerability, we must find at least 2 transactions with the same contract that are included in the same block. Furthermore, to exploit this vulnerability, there must be manipulation of the contract's storage. A contract's storage is a permanent storage in the form of a very large array that can store any important information that is used in the contract's code, for example local variables, structs, etc. The EVM instructions that manipulate the contract storage are SLOAD (which receives a key and returns a value, such that value = storage[key]) and SSTORE (which receives a key and a value and executes storage[key] = value). We record the facts tx\_sload(b, i, k) and tx\_sstore(b, i, k) whenever one of the previous instructions appear, where b is the block number, i is the index of the transaction in the block and k is the storage key being accessed. Our query for finding transaction order dependencies looks for 2 transactions that are at the same block such that one writes into the storage with a certain key and the other transaction reading the same key. We notice that this analysis can have a problem. To definitively say that this vulnerability can be exploited is only if we have the knowledge about how the storage is handled in the specific contract, there for resulting in false positives.
5. **Integer Overflow:** Since the EVM does not record the type of variables, meaning there is no explicit information about the original type of a variable at the compilation level. To check for integer overflow we must go over the EVM code in 2 passes. The first pass we try to recover the sign and size of different values on the stack. This is done by looking for any SIGNEXTEND instructions ( integers can be saved as 8bit, 16bit up to 256 bit, so what this instruction does is take a value and extends it to a bigger representation of the same value while making sure that the sign is the same) and SDIV ( integer division) so we can be sure what the sign of these values are and mark them as facts is\_signed(v). We also know what the size of the values that SIGNEXTEND are. The writers also assume that any value that is not explicitly marked as signed to be unsigned. To retrieve the size of unsigned values the writers exploit a certain EVM behavior which is inserting an AND instruction to "cast" the unsigned integers to their correct value. For example, uint8 value will compile to AND value 0xff (which is 11111111 in binary). Variables size are stored as facts size(v, n) where v is the variable that has n bits. In the second phase, we use the inferred\_signed(v) and inferred\_size(v, n) rules to retrieve information about the current variable. When we cannot tell what the size of the integer is, we over approximate it to 256 bits, the size of the EVM word. Using this information, we compute the expected value for all the arithmetic instructions (ADD, MUL etc.), as well as the actual result computed by the EVM and store them as facts. Finally, we use the Integer overflow query to check if an overflow has occurred, meaning we check if the expected result equals to the actual result. We notice that this analysis can have a problem. For example, if a contract contains code that yields AND value 0xff but value is a uint32, the algorithm will mark the variable as uint8, which can cause both false positives and false negatives. However, this type of issue is usually rare because it means that the developer that wrote the Solidity code of the contract has used bit manipulation which is usually very rare.
6. **Unrestricted Action:** The type of vulnerabilities that the writers chose to check are unrestricted Ether transfer using CALL, unrestricted writes using SSTORE and code injections using DELEGATECALL(calls a method in another contract using the storage of the caller contract and the caller's msg.sender and msg.value) or CALLCODE. We notice that before a restricted action there should be an execution trace before the restricted operation which conditionally jumps depending on the caller. There for, it is easy to check the SELFDESTRUCT case, especially because the caller address cannot be forged. For SSTORE case, we track whether the message sender influences the storage key or the address to call. For the code injection case we check whether the passed data influences the address called by DELEGATECALL or CALLCODE. NEED TO FINISH

Analysis Results

In this section, the writers' goal is to show how much of the money is at risk. The total money is 3 million Ether, or 600 million dollars. This is done by first running the tool they developed and find the total of Ether at risk which will later be used as the upper bound across all vulnerabilities. The second step is manual analysis of the reported contracts at risk which have the most Ether at risk for each vulnerability type.

1. **Re-Entrancy:** 4,337 contracts have been flagged as vulnerable out of 457,073 transactions by the other studies. After running their own tool, only 116 contracts have been found vulnerable to this attack. The total amount of Ether that is exploitable is 6,076 Ether or 1,200,000 dollars. Manual analysis of the top contracts discovers 5881 Ether was concentrated in one account which was already found vulnerable by the authors of the contract itself and performed the attack themselves to confirm this.
2. **Unhandled Exceptions:** 11,427 contracts have been flagged as vulnerable out of 3.4 million transactions by the other studies. After running their own tool, only 264 contracts have been found vulnerable to this attack. The total Ether that is at risk is calculated by: the minimum value of the balance of the contract at the time of the unhandled exception and the total Ether that failed to be sent, and we sum this up across all contracts to get a total upper bound. By this calculation, 271.89 Ether is at risk for this vulnerability. Manual analysis of top contracts confirms that in all cases the issue came from a misuse of a low-level Solidity function such as send.
3. **Locked Ether:** 7,285 contracts have been flagged as vulnerable by the other studies, with a total worth of 1.4 million Ether or 200 million dollars. After running their own tool, the writers didn’t find any vulnerable contracts. The writers conclude that the dataset that was used simply doesn’t have a contract with the DECUNSTRUCT locked ether vulnerability. For a sanity check, the writers retrieve the contracts that were affected by the Parit Wallet bug and get the same results as other studies have shown, with a total of 306,276 Ether locked.
4. **Transaction Order Dependency:** 1,887 contracts have been flagged as vulnerable out of 3,003,304 transactions by the other studies. After running their own tool, the writers found only 54 contracts potentially affected by this vulnerability. The total Ether that is at risk is calculated by summing up the total Ether that was sent resulting in a total of 297.2 Ether at risk of this vulnerability. Manual analysis was done by finding a block for each contract where this vulnerability has the highest risk of Ether. It was confirmed that at each contract it was possible for a user to read and write to the storage location within a single block.
5. **Integer Overflow:** 2,472 contracts have been flagged as vulnerable out of 1.2 million transactions by the other studies. Since the writers' tool depends on the Solidity compiler, the writers were only able to recover 945 contracts with their Solidity source code. After running their own tool, only 62 contracts were found that integer overflow might have occurred. The total Ether at stake is calculated by analysis of all the transactions which resulted in integer overflows and summing up the Ether at stake. The writers approximate that there is 1,842 Ether at stake, which is most likely an over-approximation.
6. **Unrestricted Action:** 5,163 contracts have been flagged as vulnerable out of 3,871,770 transactions by the other studies. After running their own tool, only 42 contracts have been flagged as vulnerable to this attack, all of them were all non-restricted self-destructs which held none of them held Ether at the time. **NEED TO FINISH**

Table

Description automatically generated

Limitations

1. Soundness vs Completeness: **NEED TO FINISH**
2. Dataset: Running the new tool on only their dataset means that the writers might be missing some exploits which actually occurred, for example contracts that were affected by TheDAO and the Parity wallet bugs. However, one of the main goals of this paper is to quantify what fraction of vulnerabilities discovered by analysis tools is exploited in practice and the writers' approach does exactly that.
3. Other types of attacks: This tool and paper do not cover other attacks besides the 6 that were discussed . A decision to the scope of the tool had to be made and there for the writers to focus on the vulnerabilities which were most discussed in the literature.

Discussion

Even with the limitations of the study, the exploitation of smart contracts is vastly lower than what was thought. The factors for this are:

* Distribution of Ether among contracts: only 2,000 out of 23,327 contracts had actual Ether, which most of them had no more than 1 Ether. The top 10 contracts hold 95% of the total Ether. This shows that as long as the top contracts cannot be exploited, the total amount of Ether that is actually at stake will be nowhere close to the upper bound of "vulnerable" Ether. This fact generalizes with to the whole Ethereum blockchain and not only in the dataset, such that out 15.5 million contracts, only 0.5 million have Ether. The top 10, 100, 1000 accounts respectively hold 54%, 92% and 99% of the total amount of Ether among contracts.
* Manual inspection of top contracts: Inspection of the top 6 contracts marked from all the tools in the dataset seemed quite secure and the vulnerabilities flagged were defiantly not exploitable.

Conclusion

In this paper, we surveyed the 23,327 vulnerable contracts reported by six recent academic projects. We proposed a Datalog-based formulation for performing analysis over EVM execution traces and used it to analyze a total of more than 20 million transactions executed by these contracts. We found that at most 463 out of 23,327 contracts have been subject to exploits but that at most 8,487 ETH (1.7 million USD), or only 0.27% of the 3 million ETH (600 million USD) potentially at risk, was exploited. Finally, we found that a majority of Ether is held by only a small number of contracts and that the vulnerabilities reported on these are either false positives or not exploitable in practice, thus providing a reasonable explanation for our results.

More stuff

The main difference between the way this paper analyzes the contracts and other papers is that other papers mostly build a static code analysis tools, which only reviews the contract code itself and flagging their vulnerabilities, some of these tools even check more vulnerabilities than this paper's tool. On the other hand, this paper's tool analyzes each "vulnerable" contract that was flagged by the other tools created by the other papers, and checks whether these vulnerabilities have actually been exploited by following the trace of transactions that were all recorded to the Eretheum blockchain. This is done by gathering "facts" and placing "rules" and "queries" such that if the queries are true, then this means that the specific contract that was checked might have been exploited with a certain vulnerability.

Our logic is sound if when we can show something is true with a proof tree which leads to syntactic entailment then that proposition is also true in the semantics

כלומר אם הצלחנו לבנות הוכחה כך שלבסוף קיבלנו איזושהי מסקנה מתוך הסינטקס עצמו, אז גם מבחינה סמנטית המסקנה עצמה היא נכונה

If something is true in the semantics, that is if we have something that is true in real life then we can show it is true using the syntax

Ethereum concepts:

Decentralization: The data is recorded and stored on multiple devices and no one entity controls the data record or storage process

Transparency: The way in which transactions are recorded on a ledger that is available for everyone to see and that is saved on a network of computers around the world

Immutability: The data that is stored on the blockchain cannot be changed, forged or altered thanks to cryptography and the blockchain hashing process

Critique

* Code examples for each attack.
* Explanation of relevant instructions and how they work.
* Obscure Facts and Rules
* The tool relies on that the contract was written in Solidity before compiled in order to find exploited vulnerabilities such as Integer Overflow and Unhandled Exceptions

Unsigned: an int that cannot be negative, and all 32 bits are for positive numbers

Signed: an int that can be both positive and negative. Represents less positive numbers than an unsigned int