# Vulnerabilities Checker

STEFAN KOEHN, University of Applied Science, Germany

## 1 INTRODUCTION

In this paper I would explain the tool to check possible vulnerability on a smart contract of a Ethereum block chain. First, I explain short what a Blockchain is, what we do with a smart contract and after all the well-known vulnerability of a smart contract written in the language Solidity.

### 1.1 What is the Problem?

If you write a smart contract with the program language solidity, you have no nice IDE, which no notice that you build a back door or faulty code, which can be exploited later. Such exploits come not only from the programmer. Some are bad design of the language or faulty behavior of datatype like int or double. Bad implementation is a big point of exploitable code. Developer of smart contracts ignore best practices or do code with comments like "To-do fix later" and this is too dangerous of the live time of the code.

### 1.2 How to fix that problems?

First, a fix whereas developer to study more often or better the program language documentation or go on pre-training of best practices. Another way is to rework old code or write test and let check this from another developer or quality expert. Finally, a team can develop a quality gate in a dev ops program, like Git hub, Azure DevOps or Bitbuckets.

But this are human mistakes, what about bad implementation of the language? Trivial implementation faults which come later in the stadium of the language?

In a medium article of the year 2018, the author vasa show 16 of such exploits on a smart contract with some tips and trick to prevent this mistake.

In this article the show implementation mistakes like over- and underflows of datatype with mathematical operations. Here is the solution approach to overwrite these operations with your own or to implement a new library which contains this operation. Otherwise, he recommends using a new library which done this work and implements "save maths operations".

Another exploit called Entropy illusion. This means the developer of a contract tries to create an entropy for random calculation on a blockchain. The key problem is that all calculation and operation on a blockchain are static. So, there are no randomness on a blockchain and so in a contract. So, if you create a gambling game and try to implement randomness in this game, do not forget that the "game" is still a smart contract on the blockchain, a miner which solve the block control the pseudo randomness. That mean if he does not publish it chain and try to solve another block to get the right hash to win all the games. The Solution for that problem is, to get another random source as the blockchain. That can be some other server or system, which generate entropy for you. So, you can use this new randomness for the contract and the miner have not the full control of it.

### 1.3 How I try to fix such problems

In the research of this exploit I develop a chrome extension, which can check solidity code on GitHub via one or two clicks. The Algorithm I wrote can migrate in other IDE and so your IDE can warn another developer which create new code.

Author's address: Stefan Koehn, stefan.koehn2@stud.hs-flensburg.de, University of Applied Science, Flensburg, Germany.

The actual version of this tool is currently a develop status. Key functionality like scan code without click or collect multiple source code file are work in progress. The extension is written in vanilla JavaScript and the chrome extension functions. Implement exploits scans are the over- and underflow and the timestamp attack. Here later more.

The next "great" feature should be a re-entry attack scanner. I do not implement the feature in the extension but have some ideas how it could be implemented and a theoretical algorithm to check such type of attacks.

## 2 THE CHROME EXTENSION

The extension split in three pieces.

- Implementation the backbone to use it in chrome.
- the scanner and validation
- the algorithm of the validation

The program language is here Javascript, most of them vanilla, that mean with no framework or other libraries. The only lib that here finds use, was the lib to write and use a chrome extension.

Overall, the extension is a big string compare tool, which take and get code from GitHub and save this in the local storage of the browser. This mean if you close the browser no results are saved in an extra file.

The advantage of this is, the user does not need a separate persistent place, like directory on the computer or a database or a cloud connection.

### 2.1 Client-side

Here we find four files:

- Background.js
- Popup.js (name is work in progress)
- Popup.html
- Manifest.json

*2.1.1 Background.js.* Here is the logic of the chrome extension. If you want to add or manipulate the tabs or current pages of the google chrome browser, you need to add that here.

In the current state it implements a push notification, an alert, if the user installs the extension correctly. Another implementation is that the extension only reacts on the github.com page. That means if you click the button on other sites like google.com the extension does nothing.

If you want to change the reaction url, you must do it here under the conditions: hostEquals: `"TYPE_URL_HERE"`.

*2.1.2 Popup.js.* Popup.js, the name is work in progress, is the heart of the application. The Algorithm, the verify methods and the data collect functionality are implemented here.

Above I call the extension a string compare tool. That is. The popup.js is like a simple state machine. After collect the data of the current page, in the current state unfortunately you can collect only from the raw.github page and only one "file", the extension load the file from the local storage and parse the string in a readable state. In the different methods, the tools search for the exploit's key words and structures. If the tool finds one of these, it increments a counter. The next idea is highlight the faulty code in the raw.github file.

Another feature is to detect re-entry attacks. Unfortunately, this is not implemented now, and I have only a theoretical draft of this. More in the Algorithm section.

*2.1.3 Popup.html.* This file is the view file of the extension. In this extension, I implement a very simple and basic UI with two buttons, and one text field. If you want do fancy stuff and animation of the view, you can do it here and add this.

*2.1.4 Manifest.json.* The manifest is the configuration file of the extension. In this file you can and must declare which permissions the extensions have and can uses. Furthermore, the name, version, description e.g are written here.

An interesting part of this config file is the flag `"page_action"` and background. These flags call the extension, which javascript file and html files it must load for the view and logic.

## 2.2 Algorithm

The Algorithm I use, was built from my research and the medium article of vasa.

In the extension are implement two of these exploits. This are trivial exploits with simple fixes and detection keys. The first exploit is the arithmetic over- and underflow the second one is the block timestamp manipulation.

*2.2.1 Arithmetic over- and underflow.* After the extension collect and parse the source code, the method checks if the developer uses an extern "save math lib", if it used, this validation will be skipped.

In the next step the tool search for integers, common ones are int8, int16 or int 32. Is there one of this, the next step is to check if the variable is used in a mathematical operation like addition or subtraction. Hit the tool some of these operations a warning counter will be increment. An improvement of these search Algorithm is, to check if this mathematical operation is an under or overflow. Difficult at this part is that the operation is linked with a wallet call.

So, we do not know the balance of these wallet and variable, so we cannot calculate an over- or underflow.

*2.2.2 Block timestamp manipulation.* The next Exploit is a Block timestamp manipulation. This faulty code is like a code smell. The developer tries so creating a randomness with the block timestamp. The problem is that the miner can after solving this block, manipulate the timestamp with an offset of a few minutes. An example is a gamble game where the send ether locked in the contract for a timespan.

So, the miner can "deadlock" this because his timestamp is in the future of a few minutes.

The algorithm here is a little bit trivial. The tool checks the string of the block.timestamp call and the alias "now". If it hit, it will be checked if the timestamp was used to create randomness or for a lock functionality. If it true, the error counter get increment. Here we can build an improvement that the faulty code line gets highlighted in the UI, so the user can see the mistake.

## 2.3 Theoretical draft

In this section I will explain the features that will be planed but are not in the extension.

*2.3.1 Re-entry Attacks.* This kind of attack abused the feature of a smart contract to call und user another external contract.

So, an evil developer can write a malicious smart contract and can use it as entry point of a call of an external contract, so this transaction can be hijacked from the malicious contract via fallback functions or other transaction code to call back into itself.

The harmful part can be a payment with an amount of ether that the evil developer transacts to himself or to destroy the other smart contract. Another action could be that the transaction cannot be finished so nobody gets the ether and then the ether is lost.

My solution of this is a combination of facts that other smart contract developer got with there work with a contract.

First, we need a contract. So, after we got a contract the script checked if it calls an external contract. If this true, we must search in the repository for the external call. If we cannot find them, an error counter will be incremented, or we directly throw a warning or error message. After we found the other smart contract, we iterate over it and check if it calls an external contract to.

We iterate over the contracts so long until no external call action is executed.

After that we found the last entry point, we search in the contracts for buzz words or typical logic failure.

In the article of vasa he explain three prevent technique. The first is to search of own implementation of a transfer function. Here we need to use the built-in version, because the transfer function needs 2300 gas to call external function. That is not enough to call many external contracts and the transaction failed. If we found an own transfer function the counter will be increment and a tip appears after the check to inform the developer that he should use the built-in transfer with the above reasons.

The second technique is that the state changed variable execute before we call an external contract or send ether. The reason is that the external contract will not manipulate the state of these variable.

The last technique is to block the contract with a variable, state, or other mechanism if we call an external contract.

So, if the contract its locked, we cannot execute a re-entry. The contract gets unlocked If all external call is done and we are in our contract to finish the transaction.

*2.3.2 Entropy Illusion.* What a Entopy Illusion is I explain above How to fix that problem.

My idea is so extent the block time stamp check algorithm with functionality that can recognize if the developer creates its own randomness on a blockchain or check if the developer does not use an external Api call or something like this, to get an entropy. Here we throw a warning and example what he can do and the reason, why it is bad code style to create his own randomness.

## 3 CONCLUSION

In my study of this topic, I often ask myself, how to solve such elementary problems, like over and underflows, uses of “fake randomness” or using of transaction problems.

I think many of these problems could be fixed with patches of the language Solidity or write extensions for it like the external math lib, which overwrite the old mathematical functions. Other fixes could be to rewrite these methods and catch this kind of error.

Another implementation could be a true randomness function. So, nobody needs to use his own implementation with connection of the blockchain.

In the IDE we can use better code fixes and better checker that recognize early like in the implementation phase and give the developer hints and tips to improve the code and fill security issues and prevent the contracts of attacks of faulty code style or self-written malicious code.