Nathan Beer
Smart Contract Engineer Intern

Whitepaper

Why and How to test Smart Contracts?

Blockchain has many advantages, thanks to the underlying technology which allows, in particular, ***the emergence of Smart Contracts***: the automated and clever contracts.
These contracts, **although secured** by Blockchain, can ***have errors hidden within the source code***, and in some cases, something even worse… hidden ***vulnerabilitie***s.

q-leap
software quality experts

Purpose

Blockchain and Smart Contracts have introduced new concepts, but most people don't properly understand the consequences of these concepts in terms of application quality.

In addition, most people are not aware of what and how to test smart contracts.

This Whitepaper has different objectives

The primary objective is to demonstrate why it is crucial for Smart Contracts to be tested. We will explain why the contracts must be tested and show the different consequences of not doing so.

Secondary objectives are to explain the different possibilities that are available to test a contract, the different tools that we can use to conduct the tests, the different ways in which we can automate these tests, and last but not least, to explain how we came to realize the various tests.

Background

In order to explain how and why we must thoroughly test contracts, we must explain and introduce Blockchain and one of its principal features: The Smart Contract.

What is Blockchain?

Blockchain is a **decentralized**, **distributed** and **public** *digital ledger* (based on DLT) **used to record transactions** across many computers.

In other words, it is a public and distributed database. But **depending on how it is governed,** Blockchain can be either ***public*** or ***limited*** or ***private***.

- ➔ Within a public Blockchain, ***everyone can*** read and write.
- ➔ Within a limited Blockchain, ***everyone can*** read but **only a central agency** can write.
- ➔ Within a private Blockchain, **only a central agency** can read and write

The advantages of Blockchain are as follows:

- ➔ Complete visibility and tracing of the information

  It allows one to see all the *executed transactions (exchange of information) since the beginning*. As a result, anyone who has access to the Blockchain can read the information stored within the transactions. **Any information stored within the Blockchain is open and cannot be hidden**.

- ➔ Decentralized system

  Blockchain is **based on DLT** (Decentralized Ledger Technologies). In other words,, Blockchain *isn't based on a centralized system* (such as well identified servers) **but on a P2P network of nodes**, ***and as a result, all the data is distributed*** on different nodes called "workers" or "miners". Each node corresponds to the different users of Blockchain.

- ➔ Complete Trust and Security

  Once **information** is stored in the Blockchain, it ***can neither be modified nor deleted***. The transaction (which stores the data) is stored within blocks, which are **validated by all the nodes** of the networks. As such, the record cannot be altered retroactively without the alteration of all subsequent blocks and the consensus of the network.

  The validation of transactions can be carried out in different ways, and it depends on the Blockchain. There are two principal ways this is done: the proof of work and the proof of stake. ***Proof of work*** is more commonly used, and is further explained below:

Proof of Work:

This method ***uses a mathematical problem*** which, when solved, shows "proof" that the work has been finished by a *worker*.

In order to solve the problem*, it must have important computing power*, provided by the "workers".

The security of this method may be considered as irrefutable. ***If the computing power of a group of malicious workers*** doesn't represent the absolute majority of the computing power, **then modifying it is impossible**.

In addition, the more workers there are, the more difficult it becomes to obtain an absolute majority.

What is a Smart Contract?

One of the great features introduced by Blockchain technology is the **Smart Contract**.

Smart Contracts are **electronic contracts**. They are *automated*, *clever*, and **they allow us to establish a contract** between different parties in an automated and trusted fashion.

> *For example, it can establish a sell between two people or allow an auction between many people.*

A smart ***contract is essentially a script*** that is generally developed using the **Solidity** programming language, that *presents the conditions required to proceed with its execution*.

In order to use a contract, **we need to deploy it**. We can choose to either deploy it **from a new address** or from **an existing address**.

In the Ethereum blockchain, an address is used to identify a contract or a party, and it should be noted that all addresses are unique.

Nonetheless, in order to deploy a contract, a binary file as well as the interface file (generally a abi file or a json file) are both required. In order however to **get the binary and interface files**, we need to compile the contract (the **Solidity** file), and in order to compile a contract, we must use the solidity compiler (named "Solc"). This compiler requires python to work.

In order to prove the irrefutable trust of this system, users who wish to use the contract must deploy it, keeping in mind that the initials conditions (represented by the source code) and the functions are known. As a consequence, a user who wishes to use a contract is forced to know the initial conditions because he/she must deploy the contract before being able to use it.

Smart Contracts offer many advantages:

- ➔ **Reduction of fees**: The fact that smart contracts are automated and secured allows one to execute a transaction **without a trusted third party** who would otherwise require payment for their services.

- ➔ **Reinforced Security:** Thanks to the technology behind Blockchain, all data stored within the blockchain and used by the contracts cannot be deleted.

- ➔ **Liberties**: the fact that contracts are based on the "Turing Completeness" concept allows us to take advantage of the numerous actions offered as a result. For example, we can create many different contracts with the help of the Solidity programming language, such as an auction, a bank, a currency, etc.

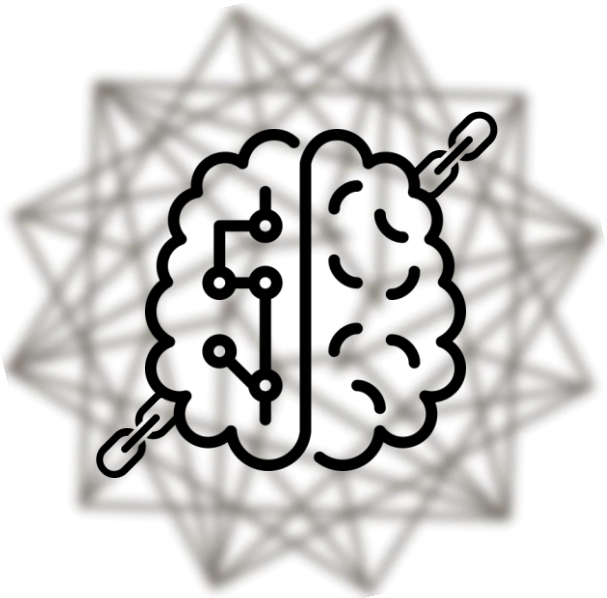
Smart Contracts are ***deployed and stored within the Blockchain***. In this way, once a contract is deployed, *it cannot be modified*. The fact that contracts are developed by humans increases the chances for bugs and vulnerabilities to be introduced as a result.

If a contract has errors in its source code, we can re-deploy a new version of it, but the old version(s) still exist in the Blockchain and everyone that possesses their address(es) can use them.

It is important to note that a contract can have vulnerabilities and faults which can be used to exploit it in a fraudulent way.

> ***For example**, we can imagine that a contract in the financial domain can generate significant losses of money if it has vulnerabilities. In a medical domain, the consequences can be even more serious…*

It is therefore imperative to confirm the ***proper functioning*** of the contracts and to ***eliminate all the vulnerabilities*** within them. This can be accomplished by Testing the contracts with the help of the different testing tools and methods that are available to us.



First approach

Why test a contract and what are the different tools and methods available to do this properly?

In this part, we will speak about the different tools and methods to test a contract. In addition, we will also go a step further and point out the advantages and disadvantages for each of the respective tools.

What are the existing types of tests?

In the testing sector of the IT domain, there are different kinds of tests such as:

Unit Tests

➔ The purpose of these tests is ***to verify the functioning of the source code***. These tests check that the functions of the code return the expected results.

➔ To do these tests, we must generally create files with test-code (which can be developed in different languages, e.g., JavaScript).

Vulnerability Scans

➔ With these tests, the goal is ***to check if the source code has vulnerabilities***. It's important to always be aware that source code that is error-free could potentially still contain vulnerabilities.

➔ *Vulnerabilities are generally found in the structure of the source code.*

Code-Coverage Tests

➔ These tests *are linked with Unit Tests*. These are responsible for ***checking if the test-code covers ALL parts of the source code*.**

➔ While it's good practice to create unit tests, it should be noted that these **tests must cover the entire source code** if we want them to be truly efficient.

Performance Tests

➔ These tests allow us ***to check the latency and the efficiency*** of an application.

How can we implement these tests on Smart Contracts?

The types of tests listed in the preceding section **can all be applied to Smart Contracts**. There are different tools available to assist with this. A majority of the time, these tools are developed using either JavaScript and/or Python languages.

*In order **to execute JavaScript and Python tools**, the following are required:*

Node.js

➔ It can execute JavaScript scripts and also provides many useful features
➔ *For more information, please refer to the following link: https://nodejs.org/en/*

Python

➔ It allows us to execute python scripts (.py files for example) and python-based programs.
➔ *For more information, please refer to the following link*: https://www.python.org/

Unit Testing with Smart Contracts

Unit tests can ***easily be executed against contracts***: There ***are many different frameworks*** that allow us to accomplish this:

Name	Pros and Cons	Popularity
Embark	**Pros** - **Solid framework** that allows you to apply the TDD. - Uses Mocha library. - Allows you to automatically deploy contracts and DApps. - Homepage: https://github.com/embark-framework/embark **Cons** - Doesn't use Chai library.	★★★
Truffle	**Pros** - **Development environment** and **testing framework** for Ethereum contracts. - It uses *Mocha* and *Chai* libraries - It allows you to compile, deploy and test the Smart Contracts, and in addition, configure the deployment scripts. - Provides an Interactive console. - Comes with **Ganache program** which can *create and manage* a personal and *private Blockchain*. - It also **provides a CLI program** to create a test Blockchain named "testrpc-sc". - Homepage: https://truffleframework.com/ **Cons** - None	★★★★★

There are many other frameworks and tools like “Populus” and “Dapple”, but the most commonly used frameworks are “Truffle” and “Embark”.

We selected to use **Truffle** to perform all the different unit tests for the following reasons:

- It is ***relatively simple*** to use and configure
- It ***can be used with Ganache*** and offers full compatibility
- It permits us to write the test code using *JavaScript* whilst utilizing the Mocha library
- It provides the ability to compile and deploy the code for testing

Vulnerability Scans on Smart Contracts

Vulnerabilities ***can generate severe consequences*** if they are not fixed. There are different available to discover the vulnerabilities present within the code:

Name	Pros and Cons	Popularity
Securify	**Pros** - Can scan a contract by simply copy-pasting code or uploading a file - Homepage: https://tool.smartdec.net/ **Cons** - Can only be used with an internet browser. - Simple to use but difficult to automate.	★★★
SmartCheck	**Pros** - Can scan a contract by simply copy-pasting code - Homepage: https://securify.chainsecurity.com/ **Cons** - Can only be used with an internet browser. - Simple to use but difficult to automate.	★★
Slither	**Pros** - **Static analysis framework** with detectors for many common Solidity issues. - Easy to use. - Possibility to export results in JSON format. - Possibility to automate. - Provides many useful options. - Homepage: https://github.com/trailofbits/slither **Cons** - None	★★★★★
Mythril	**Pros** - **Very powerful** analysis tool. - It uses *concolic analysis*, *taint analysis* and *control flow checking* to detect vulnerabilities. - **Complete and more efficient** than the above-mentioned tools. - Homepage: https://github.com/ConsenSys/mythril **Cons** - None	★★★★★

If you want more examples of tools, please refer to the following link: https://consensys.github.io/smart-contract-best-practices/security_tools/

We chose **Slither**.

This choice can be explained with the following reasons:

- **At first**, we chose ***Mythril***, but we encountered many problems during installation. For this reason, we chose Slither instead, which can be easily installed and configured.
- Slither is ***relatively efficient concerning the scan***, and it allows you to *export the vulnerability report in JSON* format which can be useful to parse and present

Code-Coverage on Smart Contracts

Code-Coverage is an important way to ensure that the unit tests are as effective as possible. In the Smart Contracts sector, there are two main tools to perform code-coverage:

Name	Pros and Cons	Popularity

Coveralls	**Pros**
	- Allows you to perform code-coverage verifications with many different features like:
	  · *Repository coverage statistics*
	  · *Individual File Coverage*
	  · *Notifications of coverages tests*
	  · *GitHub and Ci integration.*
	- Seamless integration with **CI (Continuous Integration) solutions like Jenkins**
	- Homepage: https://coveralls.io/
	
	Cons
	- Requires some configuration to work.
		★★★★★

Solidity-coverage	**Pros** - Very easy to use and to configure. - It is **efficient** and performs **good verifications**. - Uses Truffle - Generates a folder with .html files which contain all the coverage results - Homepage: https://github.com/sc-forks/solidity-coverage **Cons** - It doesn't export the results to JSON format. - Offers less options than Coveralls.	★★★★

For Code-Coverage, we chose **solidity-coverage** thanks to its configuration simplicity.

Results are exported in .html files which can easily be opened within a browser. **It should be noted however**, that this tool does not allow one to export the results to JSON format to be rendered differently.

Other tools we selected

For creating and managing the Blockchain

- **Truffle** provides a tool named "testrps-sc" which can easily create and manage a Blockchain using a command line interpreter.

- **Ganache** which can do the same job, but in addition to what Truffle offers, this tool provides a *graphical user interface (GUI)* that makes the job even easier. We selected this tool as a result.

For Creating Contracts

- To **create**, **deploy** and **execute** functions of a contract, we selected an online IDE for Solidity called "Remix".

- It allows us to create the ***contract using the Solidity programming language***. It can compile the code and deploy the created contract into **a virtual Blockchain** created by Remix; **an injected Blockchain** in the browser and **an external Blockchain** such as Ganache or testrpc-sc.

We also selected a few **other tools to simplify processes** as some tools required other tools or frameworks, in order to work properly.

Web3.js

- This tool is ***used to interface*** Ethereum contracts with the JavaScript language. We used this tool to *automate the deployment* of contracts.

Analysis

How to use a testing tool? How to create a contract? How to deploy it? Etc.

In this section, we will answer all these questions and provide details on the work that was done to create a contract, test it, deploy it, etc.

In addition, we will also explain how we can automate all these processes in order to obtain a tool that can compile, test and deploy a contract into a Blockchain.

Creating the contract

Before proceeding to test and deploy a contract, ***we need to create a contract***. Solidity is the official programming language to create Ethereum contracts. Indeed, the EVM (Ethereum Virtual Machine) which interprets the contract source code, ***requires the Solidity language***. Therefore, the creation of contract requires knowledge of Solidity.

Solidity is a Turing-complete language like many other computing languages (C, Java, JavaScript, Python, …) that **looks a lot like JavaScript** except that it has special commands for interacting with the Ethereum Blockchain (send Ether, get hash block, etc.).

To write the contract ***we used the Remix IDE*** to easily deploy the contract and execute the different functions within it.

In our example, we created a contract in the Real Estate sector:

> **The contract** will be able to sell the house of Person A. Person B will purchase the house and after confirming the purchase that person will be able in turn to sell the house to another person.
>
> The contract will store the owner's history of the house, and this list cannot be modified because it is stored within the Blockchain.
>
> **In addition**, to reinforce the process and to exploit the Smart Contracts' advantages, Person A must pawn the price of the house before selling. Person B must, after the purchase, confirm receipt of the house. It is only then that the seller receives his pawned money and actually gets paid by the buyer. Thanks to this procedure, the seller and the buyer are prevented from committing a fraud, or they risk losing their money. As such, risk or fraud is eliminated.

Remix provides many features and a facility to develop contracts but is currently in development phase and thus ***can have bugs***.

Once we have created the contract, we can now switch to the next step i.e. **Unit Tests**.

Unit Tests

We now have a contract previously created, but we want to test it by conducting a variety of unit tests.

We accomplished this with the use of Truffle. Before explaining how we used Truffle, we will explain how to create a personal and private Blockchain, in order to deploy and test the new contract.

In our case, we use **Ganache**, but we could have use ganache-cli (The CLI version of ganache) or also the *testrpc-sc* provided by Truffle. Ganache allows us to create and manage the Blockchain, and Truffle uses the Ganache Blockchain to effectuate all its tests (including deployment of the tests).

Once we start Ganache, ***it creates a Blockchain*** on the default port (8545) and with default accounts. You can, if you wish to do so, change these settings.

To start using Truffle, we need to create a Truffle project with the "*truffle init*" command. This creates a default project with default settings (stored into "truffle.js" and "truffle-conf.js").

We configured Truffle with the port of the ganache Blockchain, with its gas price and gas limit, and with the http protocol.

Gas is a new unit introduced by ***Ethereum*** Smart Contracts. A contract requires computing power to be used, and this power is represented by **Gas**. It should be noted that, **Gas** isn't free and costs a fraction of Ether (the Ethereum currency).

Once we have finished configuring Truffle, we created the unit test file and the migration file:

> ***The unit test*** contains **all the test code** that Truffle executes to test the contract. It must be written with either Mocha or Chai libraries and needs to be located in the folder "/tests" of the truffle project. The test file is *written in JavaScript* and **Truffle** *requires the Mocha and Chai* libraries in order to work properly.
>
> ***The migration*** contains **the instructions** (in JavaScript) **to deploy** the contract within the Blockchain. It uses Truffle functions and web3.js to deploy it. Truffle requires this tool, otherwise it can't deploy the contract.
>
> ***Another point to note is that*** all the contracts that we want to test need to be located in the "/contracts" folder of the Truffle project.

Now we can execute the tests by executing the "*truffle test"* command in the truffle folder. ***Truffle will execute all the test suites and test cases*** and export the results with the selected reporter (JSON in our case). ***JSON reporter*** exports the test results in JSON format making it easy to parse and render the way we want it to.

Truffle shows for each **test whether or not it has passed**. In case of failure, Truffle shows the line in the code of the contract that contains the error as well as the corresponding error message.

After the unit tests, we must check if our unit tests cover every aspect of the contract's code. In order to do that, we will perform Code Coverage.

Code Coverage

To perform Code Coverage, we used Solidity-coverage. This tool requires python and can be installed thanks to ***npm***. It requires a Truffle project correctly configured with contracts, unit test files and migration files.

Solidity-coverage will ***compile***, ***migrate (deploy)*** and ***test*** the contract before checking the coverage. This process **takes a little while** to be completed, so it's important to be patient whilst this is happening.

To execute Solidity-coverage, we must execute the "*solidity-coverage*" command in the project directory.

The command doesn't accept a parameter, but solidity-coverage can be configured with different settings *using a configuration file*. By default, this file doesn't exist, but we can create it in the root of the project folder with ". solcover.js" as the filename. All existing settings are explained in the official documentation.

Another important point to note is that with Windows OS, solidity-coverage experiences some issues; it can't start the testrpc-sc program (the Blockchain) (which actually creates a test Blockchain) to migrate the contracts and perform the coverage tests.

- ➔ **To fix this problem, we set the "norpc" option to false** in the solidity-coverage configuration file and then manually start the testrpc-sc program before running solidity-coverage. .

- ➔ **With a script**, we have easily automated the launch of testrpc-sc.

When Solidity-coverage is completed, it creates a "/coverage" folder with all the results in HTML format. In the CLI, it shows the different percentages covered by the unit tests.

Once we have unit tests covering at least 90% of the contract code we can then proceed with the vulnerability scan.

Vulnerability Scan

As explained earlier in this document, **we used** *Slither* to perform the vulnerabilities scan. Slither ***can be installed by npm*** in the node.js environment.

Slither is easy to use and as long as it is mentioned within the contract path, can easily export the result in JSON format using the "–json PATH" option.

Slither also *allows you to exclude certain types of vulnerabilities* with the "–exclude-name" option. For example, if a contract has a pragma vulnerability (which represents a small risk), we can ignore this vulnerability. This feature can be useful if we want to automate the process.

Once the contract has been scanned by Slither, and only if it doesn't have vulnerabilities, can we then deploy it into the test Blockchain.

Automating Deployment

In order to deploy a contract into a Blockchain, we can ***manually do it*** with the Remix IDE for example. ***But this requires manual human interaction***, and if we want to automate the process, *we must create a script to deploy the contract without human intervention*. This allows a way for the script to be reused as a module by another script.

We thus *created a script in JavaScript*. This script used the *web3.js* framework to **deploy the contract** into the test Blockchain.

To automate the deployment, we must also automate the compilation and pipe the created files to the deployment script.

> We accomplished automating this **by creating a Batch script** which compiles the contract using the Solc compiler.

The automation of compilation and deployment was accomplished by means of a Batch file with different options (input and output options for example*). It should however be noted that the automation could have also been developed by creating a JavaScript script*.

Global Automation

To automate all the processes, we created different modules in JavaScript used within a central script.

Each module automates a different type of task:

- ➔ A script to automate **unit tests**
- ➔ Another script to automate **code coverage**
- ➔ And a script to automate **vulnerabilities scan**

Besides these different modules, we also created two other modules:

- ➔ A script used **to I/O functions** and other useful functions
- ➔ Another script used to **automate the launch of the testrpc-sc** program that creates and manages our test Blockchain.

Finally, we created the **main script which calls all the modules in the correct sequence** to automate the entire process:

- ➔ Create the test Blockchain
- ➔ Compile the contract
- ➔ Run the various tasks (unit tests, code coverage and vulnerabilities scan)
- ➔ Deploy the contract
- ➔ Delete the test Blockchain

This script provides some features:

- ➔ Ability to change the network used by Truffle
- ➔ Ability to change the **gasPrice** and **gasLimit** used for the tests
- ➔ Indicate the path for the Truffle project
- ➔ **A verbose mode** to show all the results and commands used
- ➔ Possibility to **exclude certain types of vulnerabilities**
- ➔ Set the percentage limit for code coverage
- ➔ Etc.

In order to execute the script, we used Node.js. In addition, **if during execution, the script throws any exceptions** such as a unit test that did not pass or a scan that found vulnerabilities, the script shows the corresponding errors and stops the process.

This ensures that a contract is not deployed if there are any errors, vulnerabilities or low coverage level. **This script therefore permits us to reinforce the security** and the **quality of the contracts**. It should be noted however that the ability to **integrate these scripts in** a CI tool (Continuous integration) like **Jenkins** also exists, however we did not implement one.

To conclude, we now have a script that can compile, test and deploy the contract into a Blockchain with complete confidence concerning the quality and security.

Conclusion

This whitepaper explained why and how to implement different tests for Smart Contracts.

It also spoke about how to create the scripts and to automate the entire process from A to Z and thus ensure the integration with Continuous Integration (CI).

At the start of this Whitepaper, we explained the necessity to implement tests on Smart Contracts. Without a doubt, a fault in a contract can be exploited to either divert money or with other malicious intentions.

We then explained how to implement tests on the contracts. Different tools exist and provide different kinds of tests. Each type of test reinforces the quality and security of the contract and makes the contract safer to use

Besides implementing tests, we also created various scripts to automate the processes. Without a doubt, integration with CI (Continuous Integration) should be implemented, which helps developers test their contracts in a continuous fashion ensuring better quality and security.

To conclude, implementing tests is possible and highly recommended for Smart Contracts. In addition, these tests can be automated and easily integrated into a CI solution. The testing sector for Smart Contracts has a good future as grows the maturity of test solutions continue to mature.

Learn more now at
www.q-leap.eu
or contact us at
contact@q-leap.eu or (+352) **20 21 17**