UNIVERSIDADE DA BEIRA INTERIOR
Engenharia

Test-as-a-Service
Application to Security Testing

Nuno José Matos Pereira

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática
(2º ciclo de estudos)

Orientador: Prof. Doutor Simão Melo de Sousa
Co-orientador: Prof. Doutor João Paulo Fernandes

Covilhã, Outubro de 2016

Resumo

Num mundo onde cada vez mais o software tem um papel fundamental nas atividades do dia-a-dia, uma falha pode trazer consequências desagradáveis para os seus utilizadores. Como exemplo de uma falha grave, temos o caso Apple iCloud security exploit em 2014 [icl14a][icl14b], onde várias fotos de celebridades foram acedidas sem permissão. Para além de repercussões económicas e comerciais estas falhas levam à perca de confiança no software por parte dos utilizadores, levando assim à consequente procura de alternativas ao mesmo, podendo até resultar no abandono do software antigo. Para colmatar estas falhas, hoje em dia a indústria cada vez aposta mais nos testes de software para certificar-se que o software contém o mínimo de falhas possíveis antes de sair para o mercado.

Os testes de software servem para analisar o programa, nomeadamente na obtenção de bugs. Esta análise pode ser feita sem execução do programa (análise estática) ou durante a sua execução (análise dinâmica). As ferramentas de análise estática são utilizadas para verificar se existem potenciais execuções do programa que possam falhar durante a sua execução devido a eventos inesperados, isto faz com que o programa apresente um resultado incorreto ou até mesmo bloqueie. Foram estudadas algumas ferramentas de análise estática, JSFlow, JSPrime e TAJS, que analisam código JavaScript. Estas ferramentas foram modificadas para serem integradas na framework Nibiru.

O Nibiru é uma framework modular que tem como intuito ajudar na execução de testes de software. Esta utiliza uma arquitetura de micro-serviços, possibilitando o uso de múltiplas linguagens de programação nos seus módulos e tem a capacidade de possibilitar a execução dos seus módulos em várias máquinas. Até ao momento o Nibiru conta com três módulos operacionais, encontrando-se pronto para crescer com a comunidade informática, podendo esta contribuir na construção de novos módulos.

Palavras-chave

Testes de Software, Automatização de Testes, Testes de Segurança, Nibiru, QChecker, JSPrime, JSFlow, TAJS

Resumo alargado

A presente dissertação tem como objetivo o estudo da aplicabilidade de análises estáticas na execução de testes de segurança, no contexto de aplicações web suportadas em infraestruturas cloud. Para isso, contamos com uma framework que está em desenvolvimento, e temos como foco inicial os testes a requisitos não funcionais.

Hoje em dia, cada vez mais as aplicações desktop estão a migrar para aplicações web. As aplicações web normalmente solicitam e/ou armazenam dados privados dos seus utilizadores. Contudo, a manipulação destes dados deveria ser segura, não permitindo que utilizadores sem autorização consigam aceder a dados privados. No entanto, este facto não se verifica, demonstrando que a segurança é uma das grandes falhas de software. Como exemplo, temos o caso da Apple iCloud security exploit [icl14a][icl14b] em 2014, onde várias fotos de celebridades foram acedidas sem permissão.

No artigo "The ten most critical web applications security risks" da OWASP, publicado em 2013, estão demonstradas as falhas de software mais comuns e perigosas nas aplicações web. Com base na OWASP foi efetuado um estudo sobre as falhas e posteriormente foram encontradas soluções para algumas delas, através de análises estáticas.

A análise estática é um método de depuração de um programa que permite prever possíveis falhas no software, através da análise do código sem executar o programa. Este atributo permite aos criadores do software corrigir as falhas durante o seu desenvolvimento, ou seja, antes do software ser publicado. As ferramentas de análise estática podem analisar diferentes tipos de linguagens de programação, como por exemplo, C#, Java, C/C++, JavaScript. Tendo em conta o objetivo da dissertação, focamo-nos no estudo de ferramentas que analisavam apenas JavaScript, como JSFlow, JSPrime e TAJS. Estas ferramentas foram estudadas e testadas, de forma a entender se era possível a sua implementação na nossa framework.

O Nibiru é uma framework modular, o que permitiu-nos logo deduzir que seria possível adicionar as ferramentas JSFlow, JSPrime e TAJS como módulos na nossa framework. Esta framework tem como base uma arquitetura de micro-serviços e usa RESTful como comunicação entre os seus módulos. Neste momento o Nibiru conta com três módulos operacionais que já executam vários tipos de testes: JSPrime e TAJS, que são especializados em analisar código JavaScript para descobrir possíveis falhas; e Qchecker, que executa testes de acessibilidade e usabilidade em aplicações web.

Abstract

In a world where software gradually plays a key role daily, a failure may bring unpleasant consequences for its users. An example of a serious failure was the case Apple iCloud security exploit in 2014 where several private photos of celebrities have been accessed without permission[icl14a][icl14b]. Apart from economic and commercial implications, these faults lead to loss of trust in software by users, thus leading to the consequent search for an alternative and even result in leaving the old software for a new alternative. To address these shortcomings, the software industry started to use software testing to make sure that the software contains the minimum possible failures before is deployment.

Software tests are used to analyse the program, namely to search some bugs. This analysis can be done without program execution (static analysis) or during execution (dynamic analysis). Static analysis tools can be used to check for potential execution of the program that have not been prematurely aborted due to unexpected event at runtime, not ensuring that the program will display the correct result. We studied some static analysis tools, JSFlow, JSPrime and TAJS, which analyse JavaScript code. These tools have been modified so they can be integrated into the Nibiru framework.

Nibiru is a modular framework that aims to help in the implementation of software testing. It uses a micro-services architecture, enabling the use of multiple programming languages in his modules and has the ability to enable the implementation of its modules on multiple machines. So far the Nibiru has three operating modules and its ready to start growing with the community, so they can contribute in the construction of new modules or make small adjustments on the existing testing software to integrate the Nibiru framework.

Keywords

Software Testing, Testing Automation, Security Testing, Nibiru, Qchecker, JSPrime, JSFlow, TAJS

Contents

1 Introduction 1

1.1 Objective . . . 1

1.2 Document Structure . . . 2

2 From Testing to Security Testing 3

2.1 Software Testing . . . 3

2.1.1 On the need of Testing . . . 3

2.1.2 Central Concepts . . . 4

2.1.3 Principles of Testing . . . 5

2.1.4 Software Quality Assurance . . . 5

2.2 Security Testing . . . 6

2.2.1 Vulnerability Assessment . . . 7

2.2.2 Penetration Testing . . . 7

2.2.3 Run-time Testing . . . 7

2.2.4 Static Analysis . . . 8

2.2.5 Top Web Security Known Issues . . . 8

2.3 Technological Environment . . . 11

2.3.1 RESTful . . . 11

2.3.2 Micro-service Architecture . . . 12

2.4 Conclusion . . . 13

3 Static Program Analysis and Tools Review 15

3.1 Types of Static Analysis . . . 15

3.1.1 Data Flow Analysis . . . 15

3.1.2 Control Flow Analysis . . . 15

3.1.3 Abstract Interpretation . . . 16

3.1.4 Type and Effect Systems . . . 16

3.2 Static Analyses Tools . . . 16

3.2.1 JSFlow . . . 16

3.2.2 JSPrime . . . 17

3.2.3 TAJS . . . 18

3.2.4 Other Tools . . . 18

3.3 Conclusion . . . 19

4 Nibiru 21

4.1 Framework Overview . . . 21

4.2 Nibiru Front End . . . 22

4.3 Nibiru Core . . . 22

4.4 Nibiru Testing Modules . . . 23

4.4.1 Qchecker . . . 24

4.5 Nibiru TAJS Module 24
4.6 Nibiru JSPrime Module 25
4.7 Nibiru Reports Generation Module 26
4.8 Conclusion 27

5 Experimental Validation 29
5.1 Examples used to Validation 29
5.2 Module TAJS 29
5.2.1 Labyrinth Script 30
5.2.2 Captcha Script 30
5.2.3 Cryptobench Script 30
5.3 Module JSPrime 31
5.3.1 Labyrinth Script 31
5.3.2 Captcha Script 32
5.3.3 Cryptobench Script 32
5.4 Conclusion 32

6 Conclusion 35
6.1 Future Work 35

Bibliography 37

A Captcha 39

B Cryptobench 43

C Labyrinth 45

D TAJS report for Cryptobench 47

E JSPrime report for Labyrinth 49

F JSPrime report for Captcha 51

List of Figures

4.1 Overview of the System 22
4.2 Diagram of Nibiru Core 23
4.3 QChecker architecture, this figure was taken from ”Automated Analysis of Non-Functional Requirements for Web Applications” [CPSF16] 25
4.4 Module TAJS architecture 26
4.5 Module JSPrime architecture 26

5.1 Captcha Example 30
5.2 Labyrinth Example 31
5.3 Report of TAJS module testing Labyrinth script 31
5.4 Report of TAJS module testing Captcha script 32
5.5 Example of JSPrime report 32

D.1 Report of TAJS module testing Cryptobench script 47

E.1 JSPrime report for Labyrinth 49

F.1 JSPrime report for Captcha 51

Listings

A.1 Captcha Script 39
C.1 Labyrinth Script 45

Acronym

AST	Abstract Syntax Tree
CSRF	Cross-Site Request Forgery
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
JS	JavaScript
JSON	JavaScript Object Notation
OWASP	Open Web Application Security Project
Rest	Representational State Transfer
SQL	Structured Query Language
UBI	Universidade da Beira Interior
URI	Uniform Resource Identifier
WCAG	Web Content Accessibility Guidelines
XML	eXtensible Markup Language
XSS	Cross-Site Scripting

Chapter 1

Introduction

Currently most of the software that people use on the web, have security flaws that create unpleasant consequences for its users, leading the users to leave that software and find new ones. Problems like this generate bad marketing and economic problems to its companies. Applications based on web services and web application have growing security problems. Security flaws in Web applications can allow attackers to steal data, plant malicious code, or break into systems. For example in 2014 Apple had a serious security flaw, private photos stolen from celebrities iCloud accounts. iCloud accounts are designed to allow iPhone, iPad, and Mac users to synchronise images, settings, calendar information, and other data between devices. Although Apple has a security measure called two-step verification, it was bypassed using available software that allows access to iCloud back-ups [icl14a][icl14b].

To help the developers of web application, in this dissertation we introduce what is software testing, security testing and some types of static analysis. We also present an modular automated testing framework called Nibiru that executes non-functional tests, more precisely, security, accessibility and usability tests. This framework is open-source and can be used by everyone that want to execute tests on their web applications.

We contributed to the development of Nibiru framework, where we created two security testing modules that use static analysis to check if there is security flaws on JavaScript code and we also developed the architecture of communication between the modules of Nibiru.

1.1 Objective

The objective of this dissertation is to contribute to a framework that could improve the process of software testing. This framework should be automated and modular so it can keep growing, to add more functionality and more types of tests. Every user can contribute to this framework, it will be open-source and contributors can have benefits in helping the growth of the same.

In this dissertation we focused on developing the modular system of the framework and also on the development of two security testing modules, fitted in non-functional tests category. We need to combine two technologies to create this modular system, RESTful and Micro-services. At the end, the outcome should be a prototype that can be effectively used. The prototype should have some examples of interaction with multiples programing languages and types of software tests.

1.2 Document Structure

This document is divided into six chapters. On **Chapter 1**, we introduce and contextualise our dissertation, addressing motivations and objectives of this project. In the **Chapter 2**, we present the reader to the basic concepts needed to understand this project and the main technologies used in this project. **Chapter 3**, introduces the various types of static analysis and their basic concepts as well the existing tools to perform static analyses on JavaScript. The tools that we approach more deeply were tools that we analysed/tested extensively and demonstrated to be more suitable to implement them into our project. The **Chapter 4**, is the main chapter of this document where we describe our main contributions to the Nibiru so far and what is need to be done on the future to improve it. **Chapter 5**, is about experimental validation, there we demonstrate some tests that were executed and their performance. Finally on **Chapter 6**, we make the final statements about the project and some future work.

Chapter 2

From Testing to Security Testing

In this chapter we will introduce the reader to the software testing and will explain basic concepts that anyone should know to understand further chapters in this dissertation. It starts with software testing and all the concepts related. After we will introduce security tests and is problems now-a-days and in the end there are the main technologies that we will use to create the communications between the framework modules.

2.1 Software Testing

Software testing is more important than it seems. It is much more than just an execution of the software with the desired inputs and compare the expected the outputs with the real outputs. Software testing is the demonstration process that errors are present in the software. A tester should always test with assumption that there exist errors on software. These errors must be found as soon as possible in the development cycle of a software, because the later the error is found more expensive will be to correct it[Sin11].

So a good definition of software testing for a good tester should be *”Testing is the process of executing a program with the intent of finding faults”*[Sin11]. Based on this definition the testers should focus on doing test cases that have higher probability of finding faults/errors. These test cases must attack weak and critical points of the software so it can make it fail. If testers identify a failure of software those tests will be more useful and meaningful.

In conclusion testers must find critical situations on software by breaking it[Pat05].

2.1.1 On the need of Testing

As we know software testing is expensive in terms of costs and time. However launching a software without the proper testing might be more expensive and even dangerous.

Software testing is a necessary procedure because if we release a software with errors, something bad will happen in the future. If companies do not do this procedure they will have difficulties fixing the errors later, because fixing errors after release the software are way more expensive than fixing it in the development cycle. Besides that, companies will lose the trust of their clients, and they may not come back to use/buy the software because nobody wants software that is famous for not working properly.

One thing we know for sure is that software is made by humans and humans make mistakes, some mistakes are not relevant but others can be dangerous or even make the software fail/crash. Usually mistakes come from bad assumption or blind spots. This kind of mistakes are very hard to be spotted by the developer of that mistake. So, here begins the need for a team that was not involved in the development of the software, they can test it and can spot errors on software with more efficiency. This type of teams help software development companies to be more efficient on the testing processes.

2.1.2 Central Concepts

When software fails, it can just be inconvenient like a computer game that does not work properly, or it can be catastrophic, resulting in the loss of a life. In these cases, it is obvious that the software did not operate as intended. Most of the failures are simple and subtle, with many being so small that it is not always clear which ones are true failures, and which ones are not.

When we make an error during coding, we call this a *"bug"*. A software bug occurs when one or more of the following five rules is true[Sin11]:

- Software does not do something that the product specification says it should do;
- Software does something that the product specification says it should not do;
- Software does something that the product specification does not mention;
- Software does not do something that the product specification does not mention but should;
- Software is difficult to understand, hard to use, slow, or—in the software tester's eyes—will be viewed by the end user as just plain not right.

The terms fault and failure tend to imply a condition that's really severe, maybe even dangerous. A fault is the representation of an error where representation is the mode of expression such as data flow diagrams, source code, etc. This mistakes or errors are flaws that a programmer created while designing and building the software. If fault is in the source code, we call it a bug. On the other hand, a failure is the result of execution of a fault. When the expected output does not match with the observed output, we experience a failure. These defects or bugs occur because of an error in logic or in coding which results into the unpredicted or unanticipated results. A fault may lead to many failures, which can be different depending on the inputs to the program[Sin11].

Test case is a set of conditions that a tester write to see if that specific feature of the software works as expected. *Test suite* some times called validation suite, is a collection of test cases.

2.1.3 Principles of Testing

Software testing is an extremely creative and intellectually challenging task. So, there are seven principles that can be seen as basic guidelines for testing.

Testing shows presence of defects. Testing only can show defects that are in the present software. Sufficient testing reduces the likelihood of existing defects, but does not verify that no more defects exist. So after software testing, nobody can say that it is 100% defect free.

Exhaustive testing is impossible, testing everything in a program is impossible due to all possible inputs and the variety of their combination. Instead of trying to test everything, testers should focus on creating test cases that will make software fail.

Testing activities should begin as early as possible within the software life cycle and should be regular. As written before as soon as the bugs are detected, their fixes will cost less and it will be easier.

There is no equal distribution of errors in a software, during testing, it can be observed that the most of the reported defects are related to small number of modules, so if there is a bug in that module tester should focus on that module because there might be more bugs there.

The effectiveness of software testing fades over time. If you keep running the same set of tests over and over again, they do not expose new errors. Errors, remaining within untested functions may not be discovered. Anytime a fault is fixed or a new functionality added, testers need to do regression testing to make sure the new changed software has not broken any other part of the software.

Testing is context dependent. Testing a software is always different depending the context of the software. Different methodologies, techniques and types of testing is related to the type and nature of the application. Each context needs a new set of test cases.

After testing the software if we did not find any defects, it does not mean that the software is error free. Error detection and fixing does not guarantee a usable system matching the users needs and expectations. Early integration of users and rapid prototyping prevents unhappy clients.

2.1.4 Software Quality Assurance

As written before, testing is beneficial to the development cycle of a software. The goal of a software tester is to find bugs, find them as early as possible, and make sure they get fixed. If a programmer or an integrated tester carries out the duty of testing there will be bias, where errors will be missed. Just like people value independence for the unbiased and productive results it ensures, independent testing is crucial if quality of the product is required.

Software quality is defined on ISO 8402-1986 [iso86] as “The totality of features and characteristics of a product or service that bears its ability to satisfy started or implied needs”. So, following this standard there are seven key aspects that a software should have: Good design (look of the software), good functionality (executes the job as intended), reliability (acceptable level of failures), consistency, durability, good support and the last but not the least value [Pat05].

The solution for a software quality assurance rests in having a completely independent team for testing the product. There are three types of independent testing: tests executed by someone in the same team; by a team/tester from another group in the same company (for example a team dedicated to testing software); and by another external company/organisation to the developer company (for example outsourced testing). Besides software quality and no bias, independent testing ensures that the customer gets value for their hard earned money.

Our framework will fit on the last type of independent testing, because our automated framework will receive the software to be tested from other companies/organisation that want their software tested.

Independent testing is far more beneficial to a developer company because more and different defects on software can be found. An independent tester have a new set of assumptions and ideas to make the software fail, different than the software developer. On the other hand, the reports made by this testers are honest because they will not have problems with the coworkers, most of the time they don’t even know what team developed the software. Another advantage to independent testing is that a team dedicated to testing will have more experience, knowledge and formation in this areas and tools used [Sin11].

2.2 Security Testing

Security is a procedure used to protect sensitive and critical information or protect data while communicating over the network. Data must be accurate, reliable and protected against unauthorised access. Security testing is a type of non functional testing, which task is to check if a software is secure. Designing and testing software systems to ensure that they are secure is a big issue facing software developers and test specialists [Sin11] [Bur03]. Security involves various threats such as unauthorised users, malicious users, message sent to an unintended user, etc. The requirements of security to call a software secure include confidentiality, integrity, authentication, availability, authorisation and non-repudiation [Sin11].There are a few common types of security testing like vulnerability assessments, penetration testing, run-time testing and static analysis.

2.2.1 Vulnerability Assessment

A vulnerability assessment typically involves scanning for security issues using some combination of automated tools and manual assessment techniques. A vulnerability is a security hole in a piece of software, hardware or operating system that provides a potential angle to attack the system. That can be as simple as weak passwords or as complex as buffer overflows or SQL injection vulnerabilities. Security testers generally try to confirm the presence of a vulnerability without actually exploiting it. This means they don't actually execute an attack, but do their best to confirm that an attack is possible [WHA13].

2.2.2 Penetration Testing

Penetration testing, often called "pentesting" or "pen testing", is the practice of attacking your own or your clients IT systems in the same way a hacker would do. The goal is to identify security holes. Of course, you do this without actually harming the network. This kind of test is like a vulnerability assessment, except that the testers do actually exploit vulnerabilities. You often need an exploit, a small and highly specialised computer program whose only reason of being is to take advantage of a specific vulnerability and to provide access to a computer system. Exploits often deliver a payload to the target system to grant the attacker access to the system. A payload is the piece of software that lets you control a computer system after it's been exploited [WHA13].

In other words: The difference between penetration testing and hacking is whether you have the system owner's permission.

2.2.3 Run-time Testing

Run-time Testing, also referred as dynamic testing and black box testing, it is a kind of test that involves assessing the system for security issues from the perspective of an end user. The tester does not have access to source code or other detailed knowledge of system internals. This is an accurate reflection of the kind of knowledge an external attacker has. Not having access to source code limits the tester's visibility into potential security issues. Because run-time tests are often time-limited in order to control costs, they may not accurately capture the kinds of attacks a dedicated adversary can find with more time [WHA13].

2.2.4 Static Analysis

Static analysis consists in checking the source code of a programs and check if the parameters are as they should be. Depending of which static analysis we are running the parameters might switch. The software code is not executed but the tool itself is executed, and the source code is the input data to the static analysis tool [WHA13].

Static analysis offers techniques for predicting safe and computable approximations to the set of values or behaviours that can occur at run-time. Its main application is to allow compilers to generate code avoiding redundant computations, for example, reusing available results or moving invariants out of the loops, this type of data must be known before the compile time. One great compiler that can use static analyses to optimise user code is the GCC [gcc16].

2.2.5 Top Web Security Known Issues

Nowadays every one is using web applications making web security a concern worldwide. Open Web Application Security Project (OWASP) is a worldwide not-for-profit organisation that intends to improve software security [OWA16], by using is community to find bugs and promote security in software in many different ways. This subsection describes the Ten Most Critical Web Application Security Risks, OWASP published this list in 2013 [OWA13].

Injection flaws, such as SQL, are considered the most harmful risk for worldwide organisations. The threat agents could be anyone who can send untrusted data to the system, including external or internal users and administrators. The agent sends simple text-based attacks that easily exploit the syntax of the targeted interpreter. This type of attack is often found in parts of queries, commands, program arguments. Although injection flaws are easy to discover when examining code, they have a difficult detectability via testing. The impact of this attack on organisations is severe, resulting in data loss or data corruption, lack of accountability, denial of access or sometimes lead to complete host takeover. Injection flaws have a major impact on businesses reputation by affecting their data which could be stolen, modified or even deleted.

Application functions related to authentication and session management are also a dangerous attack. In most cases, this functions are not implemented correctly, allowing external or internal attackers to use leaks or flaws in the functions to compromise passwords, keys or session tokens, or to disguise their actions using other users' identities. The development of authentication and session management schemes are hard and frequently customised, thereby making the search and repair of a flaw difficult, as each implementation is unique. Once the flaw based attack is exploited, the attacker can do anything in the accounts. Administrator accounts are frequently the attackers target.

Every time an application takes untrusted data and sends it to a web browser without proper validation, we call it a **Cross-Site Scripting** (XSS). Internal or external users and administrators can send text-based attack scripts, such as data from the database that exploit the interpreter in the browser. This attack scripts can hijack user sessions, deface web sites, insert hostile content or hijack the user's browser using malware. The XSS flaws are the most prevalent web application security flaws. They can be Stored or Reflected, and each one of these can occur on the Server side or on the Client side. Server XSS flaws are easy to identify whether via testing or code analysis. On the other hand, Client XSS flaws are very difficult to detect.

Insecure direct object references occur when an authorised system user, simply changes a parameter value that exposes a reference to an internal implementation object that the user is not authorised for. This kind of flaws happen because applications don't always verify if the user is authorised for the target object, such as a file, directory or database. Testing and code analysis can easily find this flaws through manipulating parameters or showing whether authorisation is properly verified. Without an access control check or other protection, this flaws can compromise all the data referenced by an object. It's easy for an attacker to access all available data, unless object references are unpredictable.

Protection is essential to protect users and their own accounts of attackers attempts to compromise the system. Attacker accesses default accounts, unused pages, unpatched flaws or unprotected files to gain unauthorised access the system. **Security misconfiguration** is another type of system flaw, which can happen at any level of application stack, including the platform, web server, application server, database, framework and custom code. The system could be completely compromised without the user know it. For detect this flaws automated scanners are very useful. Secure settings should be defined, implemented and maintained, as defaults are often insecure. Software actualisation should be done, because updating software help to improve the security by fixing previous bugs.

The exposure of sensitive data by many web applications is another system's flaw. Data like credit cards, tax IDs and authentication credentials are not properly protected in browsers. Attackers can gain access to sensitive data and any backups of that data. They steal or modify data at rest, in transit and even in your customers' browsers, conducting to credit card fraud, identity theft or other crimes. The most common flaw is simply not encrypting sensitive data. Encryption is an extra protection but many times encryption isn't well implemented and the flawed implementation permits weak key generation and management, and weak algorithm usage. Attackers typically don't break cryptography directly, because even tough browser weakness are very common and easy to detect, they are hard to exploit on a large scale. Protection of sensitive data frequently fails compromising all customers information that should be safe.

Most web applications verify function level access rights before making that functionality visible in the user interface. Applications need to check access control on the server when each function is accessed. **Flaws in access control** can give private functionality to anonymous users or regular user access to privileged function. If access rights are not verified, attackers will be able to forge it just simply changing the URL or a parameter to access applications functions. The protection of applications do not always functions properly. On one hand, function level protection is managed via configuration, and the system is misconfigured. On the other hand, developers must include the proper code checks. The detection of access control flaws is easy, however the hardest part is identifying which URLs or functions exist to attack. Administrative functions are key targets for this type of attack.

Cross-Site Request Forgery (CSRF) is an attack that forces a logged-on user's to execute unwanted actions on a web application in which they're currently authenticated. Browsers send credentials like session cookies or IP address, enabling the attacker to create forged HTTP requests and trick a victim into submitting them via image tags, XSS or numerous other techniques. Such vulnerabilities are called CSRF flaws. After the user is authenticated, the attacker inherits the identity and privileges of the victim to perform an undesired function on victim's behalf. This fact allows attackers to create malicious web pages which generate forged requests that are indistinguishable from legitimate ones. Attackers can trick victims into performing any state changing operation the victim is authorised to perform, such as updating account details, making purchases, log-out and even log-in. CSRF flaws are easily detected by means of penetration testing or code analysis.

Libraries, frameworks and other software modules are vulnerable components that almost always run with elevated privileges. If a vulnerable component is identified and exploited with automated tools, an attack can facilitate serious data loss or server takeover. A threat agent by means of scanning or manual analysis can identify a weak component, customising it and executes the attack. Components with known vulnerabilities may compromise application defences and enable a range of possible attacks and impacts. If the component identified is deep in the application, it gets more difficult to perform the attack. Applications have these flaws because development teams don't ensure that their components/libraries are up to date. The impact of this type of flaws could range from minimal to complete host takeover and data compromise, through injection, broken access control and XSS.

Web applications frequently **redirect and forward** users to other pages and websites. Sometimes the target page is specified in an unvalidated parameter, allowing attackers to trick victims and choose the destination page. Victims are more likely to click on it, since the link is to a valid site. This redirect and forward on pages enable bypass security checks. Detecting unchecked redirects is easy. On the other hand, unchecked forwards are harder to detect, because they target internal pages. Flaws like this may tempt users to install malware or trick them into disclosing passwords or other sensitive information.

2.3 Technological Environment

We introduce in this section the relevant technological environment concepts. Knowing this concepts will be required to understand future chapters.

2.3.1 RESTful

REST (representational state transfer) defines a set of architectural principles by which you can design Web services that focus on a system's resources. RESTful is typically used to refer to web services implementing such an architecture.

RESTful is based on Hypertext Transfer Protocol (HTTP) 1.1 and Uniform Resource Identifiers (URI). RESTful use HTTP verbs (example: GET, POST) to make the communications between the server and the clients in different languages. There are five key principles in a RESTful application: everything is a resource; each resource is identifiable by a unique identifier; use HTTP methods; resources can have multiple representations; and the communication is stateless [Boj15].

The principle *everything is a resource* means that data is not represented by a file, instead it is represented in a specific format. All data is available on Internet and its format is described by a content type (example: Content-Type: application/json; Content-Type: text/html).

Each resource is identifiable by a unique identifier. This intent to make every resource accessible via URI on Internet and the URI must be unique (Similar of primary keys on Databases). Another hint is the URI should be self-descriptive and should be human-readable. It helps to reduce errors [RES08].

RESTful *use the standard HTTP methods*, there are eight methods (GET, HEAD, POST, PUT, DELETE, TRACE, OPTIONS, CONNECT) on the standard HTTP published in 1999 on RFC 2616 [Soc99]. As we will see in more details in chapter 4, Nibiru only uses two of this methods (GET, POST). GET method is used to request an existing resource and it answers 200 OK if the resource exists, the 404 Not Found answer is sent when the resource does not exist and the last possible response is 500 Internal Server Error for all another errors that can occur, on the POST method we get the responses as the GET method.

Resources can have multiple representations. Resources representations can be different than the stored ones, as long as the format of the resource is supported by RESTful. For example users can do a POST of a XML type of resource, but the server is waiting for a JSON resource and this kind of requests are valid as well [RES08].

To *communicate stateless*, the RESTful service must have atomic requests, such as modifications to a resources should be done in only one request and this request should be final or completed. This happens because once the request is executed the resource will be considered the final resource. If the modification is not final we can have another request meanwhile and that can create bugs/errors. All the incoming requests must pass by a load balance, so that service can ensure stability on server side and availability of the resources to the client side of things. As final note, this last principle is the responsibility of the programmer, because he is the one that need to make the API stateless. He must create the balance of the server and reject all the requests that are not final/complete [RES08].

2.3.2 Micro-service Architecture

Micro-service architecture, or simply micro-services, is a distinctive process of developing software systems that has grown in popularity in recent years. It's popularity came from the cloud services that are changing to this type of architecture due to is manoeuvrability, scalability and its complete integration with any programing languages. These advantages give much more freedom to all people involved in the development cycle of a software that uses this architecture. In short, the micro-service architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.

Currently software needs domain-driven design characteristics such as continuous delivery, on-demand virtualization, infrastructure automation, small autonomous teams and systems that scale. Micro-services emerged from the necessity to do a small and well done job. There is no specific way to measure the size of a service, so we know that it's small when we could not split its work anymore. It also need to be autonomous because each micro-service is a separate entity, so it could be deployed in any machine or in any platform. This can add some overhead to the service due to the communications between all the services are made via network calls but this usually is not a problem, because of the easy way of having multiple machines in different location doing its jobs. This services must work independently of each other and each service can be done in a different programing language. This make that small teams could be autonomous in its work [New15].

The benefits of using this architecture are many and varied. We can say that this architecture is a combination of distributed systems and service-oriented systems.

To summarise there is the most important properties of a Micro-Service architecture [Fow14]:

- Services are easy to replace;
- Services are organised by their capabilities;
- Services can be implemented in different programing languages, databases, hardware, environment;
- Micro-Service architecture is symmetrical rather than hierarchical;
- Naturally enforces a modular structure of the software.

Robert C. Martin's wrote that one of the most foundational principles of good design is *"Gather together those things that change for the same reason, and separate those things that change for different reasons"* [Mar09].

2.4 Conclusion

In this chapter we explained to the reader the basic concepts of software testing and and security testing, we also introduce some technologies that we used on our framework. The reader shall now have the basic knowledge of testing. Next chapter we will introduce the reader to static program analyses and tools that can execute this type of tests.

Chapter 3

Static Program Analysis and Tools Review

In this chapter we briefly describe some types of static program analyses, it consists in discovering properties automatically of a program, for all possible execution paths. Depending in what type of static analysis were used different properties will be discovered. Static analysis is an automatic method to reason about run-time properties of program code without actually executing it. We also present three static program analysis tools.

3.1 Types of Static Analysis

Static program analysis can be separated in four main approaches: Data Flow Analysis, Control Flow Analysis, Abstract Interpretation, Type and Effect Systems. Each one of this approach is explained a bit more in detail in the subsections below.

3.1.1 Data Flow Analysis

Data Flow Analysis, views a program as a graph where the nodes are elementary blocks and the edges of the graph show the flow of the data from one elementary block (node) to the next. This technique is designed to gather information about the values at each node of the program and how they change over time. As an example taint analysis is a Data Flow Analysis which consists on checking all variables which have been supplied by the user this data is called tainted data because it can be dangerous if executed without caution. After checking this variables, if they are safe variables or they were manipulated they will be called sanitised variables, because when they reach an execution they will not harm the system/software. This type of analyses help to prevent injection attacks [Nie05].

3.1.2 Control Flow Analysis

Control flow analysis determine what information lead from one elementary block to the next, this information consists in which functions can be called at various points during the creation of a program. All information collected is represented in a Control Flow Graph (CFG) where nodes are instructions of the program and the edges represents the flow of control [Nie05].

There are many similarities between data flow analysis and control flow analysis, for example, in both cases the syntactic structure of the program produce a set of constraints whose least solution is aimed. Control flow analysis constraints have a more complex structure than data flow analysis.

3.1.3 Abstract Interpretation

Abstract interpretation is a method for creating approximate semantic of programs which will be used to get information about that programs with the intent of provide sound answers to questions that developers have about the behaviour that it will have in run-time [Nie05]. It can be viewed as partial execution of a program with more information about is semantics and without computing everything.

3.1.4 Type and Effect Systems

Type system is a collection of rules assigned to variables, expressions, functions, modules, this help to reduce possibilities of getting bugs, because every time that exist a combination of types it must be checked with the rules. This system can run statically, dynamically or a combination of both, type systems can have other uses besides checking the types of data on a program, it can enable some compiler optimisations, it can generate documentation and many other uses [Nie05]. On type systems we have two different types, *Type checking* and *Type inference*, the first verify whether the program is accepted by the type system, and the other infer the type that allows a type system to accept a program.

Effect system is a formal system designed to study the effects that can be produced when executing a program. Usually this effects are side effects. It is typically an extension of a type system, this system can be used on compile time to predict the damages that can be generated when running that program.

3.2 Static Analyses Tools

Static analysis tools can be used to check that the program execution is not prematurely aborted due to unexpected run-time event. The static analysis tools can analyse different types of languages, for example, C#, Java, C/C++, JavaScript. Based on the purpose of the project, we'll focus on some tools that can analyse JavaScript code, like JSFlow, JSPrime ans TAJS. We choose these tools because they should be open-source and they are most known tools to execute static analysis on JavaScript code.

3.2.1 JSFlow

JSFlow is a static analysis tool written in JavaScript and it analyses JavaScript code, searching for security flaws. It enables in-depth information flow and can analyse programs that use third-party libraries. JSFlow can operate in the two types of information flow, explicit flow and implicit flow. All the literature distinguishes between this two types because there is no technique that can do both information flow. Using just explicit flows is equivalent to using taint analyses. This is a usefully technique that can track data from unknown sources that can reach sinks. However, the implicit flow can, for example, detect code and malware injection into ads on a web page, or detect XSS, the most known code injection flaw [Inf12].

JSFlow uses a dynamic type system for enforcing secure information flow, where its semantic model closely follows the ECMA-262 standards. It monitors how information flows on the software during its execution and when JSFlow find a flaw it injects the source with an annotation. This prevents that the same security flaw exception occur on the next run. Based on this annotation we can understand how the information flow and what generates this possible flaw [jsf14].

3.2.2 JSPrime

JavaScript programs usually run on client-side applications and this can lead to easy script injections. This type of attacks are most known as DOM-based Cross Site Scripting (XSS) attacks. The traditional tools that rely on pattern-match do not detect these vulnerabilities. XSS isn't just a problem in the browser-based applications, for example NodeJS based applications can be vulnerable to this type of attacks and in this cases the impact is more lethal because attacker can get complete control of the server. A good way to detect this type of attack is to use a static code analyser, more precisely a taint analyser.

JSPrime [jsp16b] is a source code scanner that can identify security problems (SQL injection, XSS) on JavaScript applications using taint analysis. Its source code is written in JavaScript, it uses the esprima parser. This tool was made for developers and security testers to follow the code execution order and to understand type casts. JSPrime keeps track of all the objects and variables, using control flow and data flow analysis to help detect sources and sinks. It tracks every variable and object that can help in the detection of possible entry points with script injections on the application that are being analysed. However, JSPrime only has the knowledge of pure JavaScript (ECMAScript) and it is aware of the context-base of JQuery, it cannot detect 100% of the issues of a JavaScript application. There are many sources that can reach a sink that are not detected by this tool. Some are not detected because the application use more than pure JavaScript (ECMAScript).

JSPrime also has a bad report system, because it detect the line of code that have the possible security flaws. So this make that if there no beautify code in the analyses it can say that error is in one big line of code and that is not precise. Other problem of JSPrime is that its robustness is largely dependent of esprima parser. Every flaw that esprima does have a large impact on JSPrime analysis.

3.2.3 TAJS

Type Analyser for JavaScript[taj16b] is a tool that does static analysis of JavaScript code, and can infer detailed type information using abstract interpretation. TAJS can be used to detect common programming errors and generate type information for better program comprehension. It has good precision on small and medium size programs but loses some precision on large scale programs. This is not a problem because the majority of JavaScript applications are small or medium size. TAJS detect the following errors: dead code, unreachable code, calls to functions with wrong number of arguments or with arguments with unexpected types and misuse of undefined value. This last one usually generates errors in run-time and can crash the application. It covers full JavaScript language as specified in the ECMAscript [ecm15] standard, but as everyone knows browsers do not adhere to JavaScript standards. Some browsers even provide extra functionalities and almost every browser have slightly different JavaScript interpretation, for example event systems are different on the most browsers. Besides the full coverage of ECMAscript, TAJS is flow and context sensitive, meaning that it distinguishes different program points. This make TAJS produce more precise and can generate more information about errors present in the software.

Creating a lattice structure to JavaScript has its challenges. TAJS uses an intricate lattice structure, which it's based on constant propagation for all primitive types of JavaScript. However, the usual termination requirement that the lattice should have, finite height, does not apply on TAJS. The lattice structure also includes call graph information so that it can track all the functions. TAJS uses Rhino [rhi16] as a JavaScript parser and uses a built-in technique from Monotone Framework called recency abstraction that fits perfectly on the JavaScript style.

3.2.4 Other Tools

Besides the tools described on the sections before, there are other important tools on static analysis. We will only enunciate subsequently the ones that we also tested on our work. They are ScanJS [sca16], Infer (most known as fbinfer) [inf16] and flow [flo16].

ScanJS is a web-based static code analyser for JavaScript code, however this tool is deprecated. It uses Acorn library for parsing the code and generate AST. After the AST is generated ScanJS use a list of preset scan rules or developers that can construct their own rules.

Infer is a static program analyser as the other tools mentioned before and can analyse Java, C and Objective-C code. This tool is written in OCaml. Infer is focused on tracking problems like null pointer dereferences and memory leaks which usually affect largely mobile apps. Facebook developers use this tool to search for that kind of problems in their mobile applications (Facebook, Facebook Messenger and Instagram).

Infer has some limitations, for example, it does not covers concurrency, dynamic dispatch, Android life-cycles and others. Some of this problems are general problems in the static analysis area, while others are just features that are in a to do list waiting to be implemented.

Flow is a tool that find bugs with the use of type inference, without any annotations. It's written in OCaml and scan JavaScript code. Its interpreter of ECMAScript, and it works on real-time scanning of all JavaScript files in a folder. Flow can catch the most common bugs of JavaScript like silent type conversions and null dereferences.

3.3 Conclusion

In this chapter we showed many types of static program analyses and that exists three main static analyse tools that can check security flaws on web applications. On the next chapter we will present Nibiru, our modular framework that can execute multiples software tests.

Chapter 4

Nibiru

In this chapter we will see Nibiru architecture and how it works. Nibiru is an Open-Source framework made for the industry of software development, to help in the software testing processes. Nibiru is responsible for executing the test cases and generate reports from the executions.

Framework Nibiru use micro-service architecture, making possible the use of multiple programing languages in their modules and use manoeuvrability to make possible run modules on multiple machines. Tests are executed on real machines, allowing it to improve the accuracy of the tests. It is responsible for executing tests and static analysis of multiple software areas like security, accessibility, usability.

4.1 Framework Overview

The considerations behind this framework architecture were based on specific system needs. The system requirements were scalability, performance, reliability, visibility, stability, simplicity and liveness. Based on this specific needs we used micro-service architecture that allowed to achieve the stability, simplicity and live-ness required. On other hand, we still needed to achieve scalability, performance, reliability and visibility, and to obtain these goals we used RESTful as a communication protocol between the multiple micro-services.

This framework allows the community to get involved easily by creating and extending modules on our framework, where modules can be open-source or paid, their developers can choose one of this two business strategies.

The development of the modules is easy because of the micro-service architecture of Nibiru and is protocol of communication (RESTful) as mention before. Each module only need to communicate with our framework by our API and use JSON to encapsulate data in the communication process. The figure 4.1 is a graphical representation of the Nibiru overview. The system has four essential modules: Front-end, Core, Database, Report and beside these modules the system has more modules that are called testing modules that will run the intended tests. Testing modules can be repeated as many times we want because if there is a need for a specific test, we can launch multiple modules for that specific test. At moment we have three testing modules: JSPrime, TAJS and Qchecker. JSPrime and TAJS are static program analysers, that scan JavaScript code

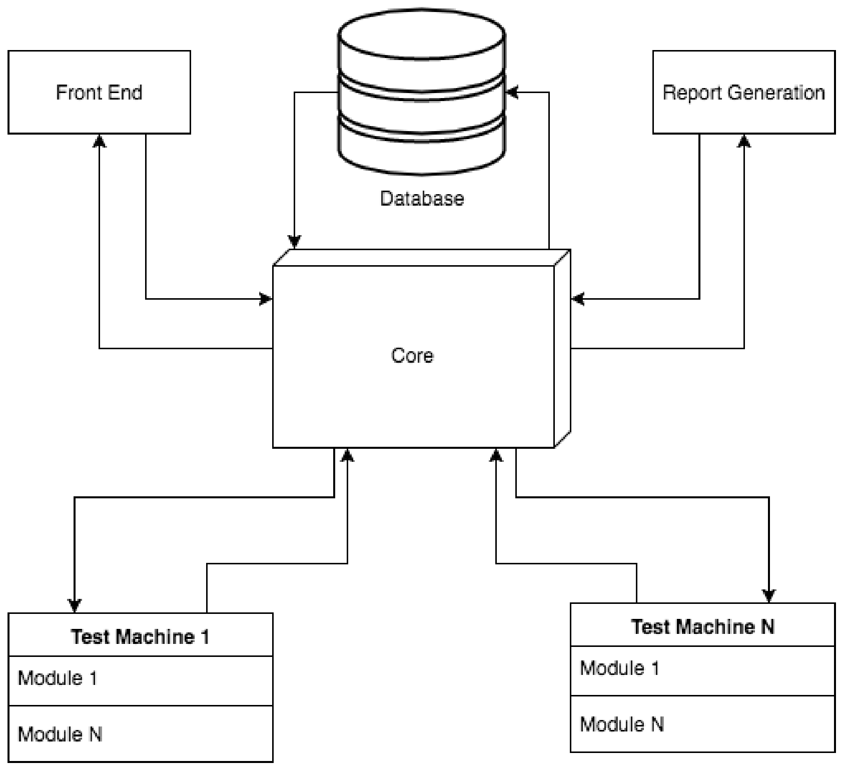


Figure 4.1: Overview of the System

for security flaws, while Qchecker executes accessibility and usability tests on web applications. All the modules will be explained in depth in the sections bellow.

4.2 Nibiru Front End

Every features of Nibiru will be available in its Front End. Users will need to pass by an authentication process to access their account, which after the authentication process is completed, the users will have access to all real machines, virtual machines, settings and features of the framework. Each type of tests may have multiple or different types of inputs and multiple settings possible, depending on each type of tests that will be executed. For example, the security modules already implemented only have a unique input, a source code (JavaScript) of the software. On settings of some tests (accessibility) the user can use Web Content Accessibility Guidelines (WCAG) or define their own set of rules, making our framework very customisable and multifaceted. Real/virtual machines have some presets configured that users can use, but they can easily setup their own custom machines to test their software as they wish.

4.3 Nibiru Core

Nibiru Core as its name suggests is the main service of our framework. It is responsible to receive and interpret information from the Front End, generating the data to send to services that must be used. Besides this, Nibiru Core does all the management and monitoring of the Nibiru modules. It monitors the load and the performance of each machine connected to the framework, and it can launch more machines if needed, so there will be no bottlenecks on the Nibiru side.

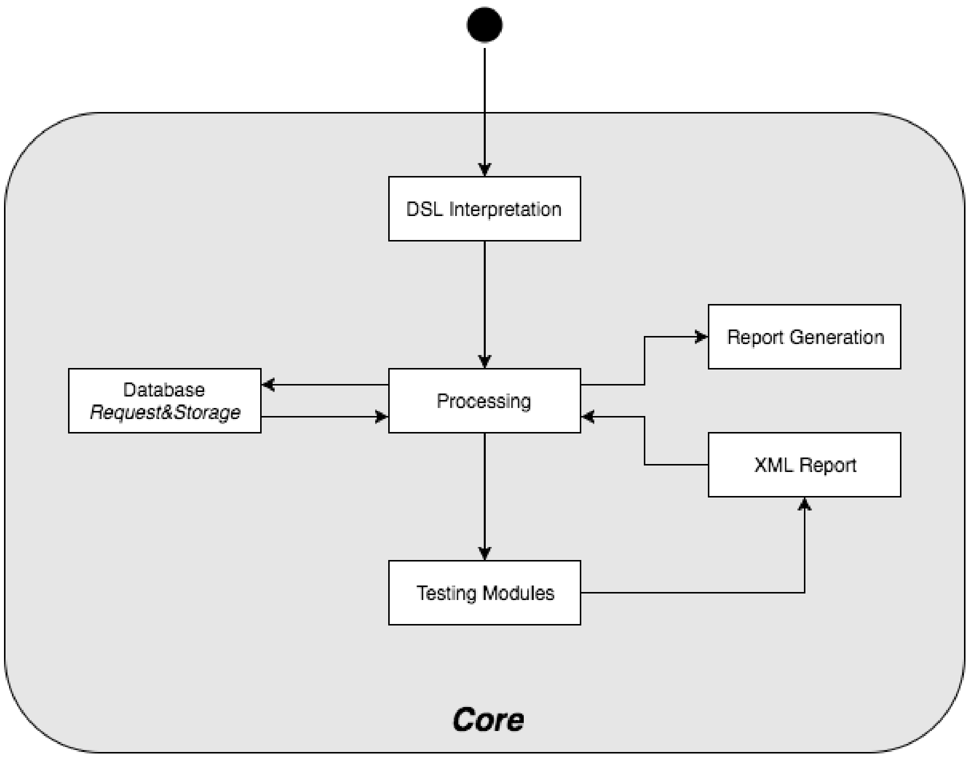


Figure 4.2: Diagram of Nibiru Core

As we can see on the figure 4.2 Core receive data from the Front End module. This data will be sent in JSON to the Nibiru Core, where the JSON information interpretation begins. The data contained within JSON are test cases, configurations to the test machines, source code to test and everything needed to test that software.

After the interpretation of the data is completed, the processing starts by saving data and requesting the ID (IP:Port/method) of every module needed to execute tests. The data is sent to test modules that will execute all tests. While tests are running, each module will produce a report on XML format, upon completion of the tests, the XML reports are merged and saved on the database. Then XML reports are sent to the Report Generation module that will convert XML reports in many other formats, such as PDF and HTML. The HTML reports are required because they are used on Front End to show the results.

4.4 Nibiru Testing Modules

Nibiru has already three developed modules, wherein two of the modules uses static analysis to verify security flaws on JavaScript. One module uses type analysis, whose can prevent users to mess with variables on JavaScript. We named this module TAJS (Type Analyser for JavaScript) because we used TAJS [taj16b] in this module and transformed TAJS to make it a micro-service application that can integrate it in our framework. The other module is JSPrime that uses taint analysis to check possible flaws in the JavaScript code like SQL injection, XSS and others. It can check the software for possible entry points to the server (variables that have interaction with the user of the software tested). As the first module we named JSPrime because we used JSPrime [jsp16b] as core for this module. Last module is Qchecker [CPSF16], which executes accessibility and usability tests on web applications. It uses standardised or custom rules to check if the web application follows that rules.

The main contributions of this dissertation are based on making modifications on TAJS and JSPrime to turn it as micro-service to demonstrate that our architecture can integrate any type of modules, as designed. The three modules are different in the type of architecture and programming languages and also their work are totally different from one to another. In the end we could make them all fit in our framework and work seamless, so everyone can fit their testing application in our framework or construct one module from the ground.

4.4.1 Qchecker

QChecker is a framework that allows automatic accessibility and usability testing of web applications. Qchecker was created by Joel Carvalho [Joe16] and we have contributed on his development as well. It can test web applications with some predefined guidelines, however users can make their own guidelines. This tool does not need the source code of the application, it can test web application only by visiting it. QChecker itself is a framework that has three main modules in his architecture, as we can see in the figure 4.3.

On top of the figure 4.3 we have the module that is responsible for integration of this tool with others systems, for example Cucumber. The second module does the rendering and manipulation of the web applications. This manipulation is needed to generate all the HTML of the web application that is being tested. After all manipulation, the generated HTML is sent to the third module, QChecker Core. This last module does specification and implementation of the guidelines. It also is responsible for executing the tests and generate the final report.

4.5 Nibiru TAJS Module

As mentioned before, module TAJS is called this way because we used TAJS static analysis tool to verify security flaws on JavaScript. Figure 4.4 introduces the TAJS Modul architecture, where Nibiru Core communicates with this module service by RESTful.

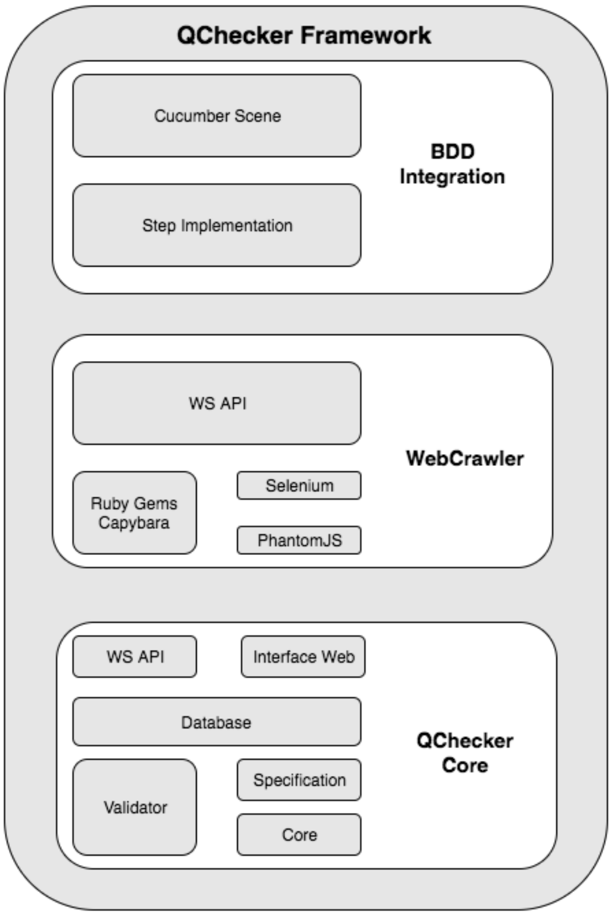


Figure 4.3: QChecker architecture, this figure was taken from ”Automated Analysis of Non-Functional Requirements for Web Applications” [CPSF16]

Nibiru only uses POST and GET methods for now, and they are the only methods that we need. To make TAJS fit in our architecture we needed to make a wrap around TAJS. This wrap were made in Python, where we used subprocess library to help us manage and launch the Java program, making the service able to receive data from Nibiru by a POST. The data come inside of a JSON, wherein the data is something like “path”:x, the x will be the path of JavaScript file. This path will be injected in the TAJS and then it starts his type analysis of that source code inside the file. When the analysis are done, the service picks up the output of TAJS and convert it into a JSON “data”:output, and send it back to Nibiru Core.

Source code of the TAJS module is available on github [TAJ16a].

4.6 Nibiru JSPrime Module

The tool JSPrime is a web page, so we needed to take his JavaScript libraries and construct a new interface to it. This new interface had a requirement, it must work with Nibiru, and so it needed to be a service. To make libraries work with Nibiru we created a RESTful service in Node.js. This service provided us communication between

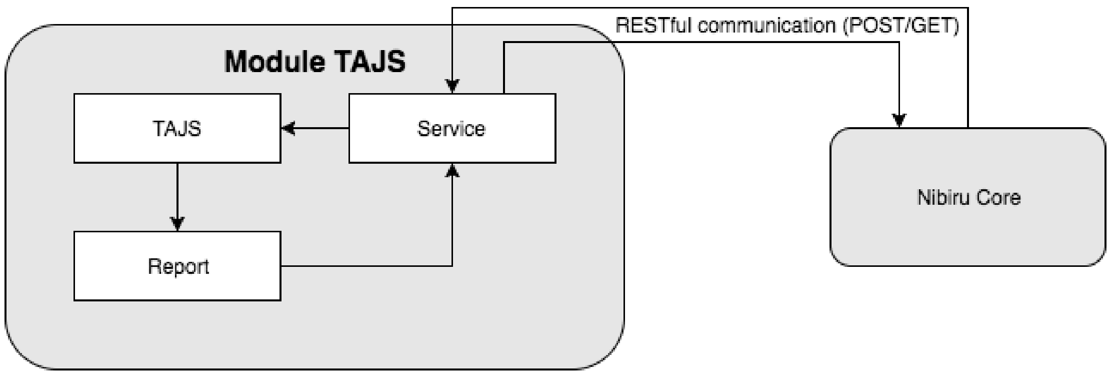


Figure 4.4: Module TAJS architecture

Nibiru and JSPrime, because with JSPrime execution inside of node.js we improved his efficiency and fixed some flaws in JSPrime, as the ambiguity of the reports where we had minified JavaScript code.

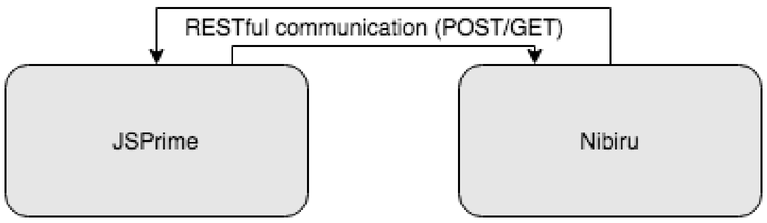


Figure 4.5: Module JSPrime architecture

As we can see in figure 4.5 the architecture of JSPrime module is very simple, because we built almost everything from the ground, helping us to simplify and optimise some processes. Similar to TAJS module, this module also receive from Nibiru a JSON that contains the source code of the JavaScript that we want to analyse. JSON package sent to JSPrime module have ”code”:x as syntax, where x is the JavaScript source code. After JSON data is received, JSPrime module does a code beautify so that we can have a readable code and an improvement on the accuracy of the reports. Then JSPrime libraries are used to execute his taint analysis and an HTML report is generated, that will be sent back to Nibiru in a JSON as usual on our modules.

Source code of the JSPrime module is available on github [JSP16a].

4.7 Nibiru Reports Generation Module

Our framework has a report module that can generate reports from the results given by the testing modules. Testing modules should give to Core an XML file with their output. Hereafter Core will merge all the XML files and the final XML file is processed on the Report module, where a report will be generated in other types of files, like PDF, HTML and others.

Since we don't have yet an unified XML tags we cannot convert the reports to another format. We are currently working on this improvement and when this is achieved we have a unified XML tag system that can generate more report file types from the same XML file. This XML file will be the only stored report in our framework and every time that the user want the report, it is generated to the file type that the user choose.

4.8 Conclusion

We conclude that Nibiru can accept almost every testing software implemented on the market. Most of them are usually ready to join our framework and others must get a few modifications to work with us, but in extreme cases every software can have a wraparound with a service that can control that program and launch his execution.

Our contribution to Nibiru is the two security modules and the architecture of communication between its modules.

Nibiru framework can be improved in several ways, but its architecture and usability are mature enough since it can evolve as quickly as users want and everyone can help in its development. On the next chapter we will show Nibiru experimental validation of some cases that we tested and its performance.

Chapter 5

Experimental Validation

This chapter will demonstrate results of three scripts that we analysed as well their experimental validation. We evaluated the same scripts in both security modules (JSPrime and TAJS modules) to compare results.

5.1 Examples used to Validation

To validate our modules we analysed more than 50 scripts. These scripts have different sizes, wherein we call small to scripts that have less than 10 lines of code, medium to scripts that have between 10 lines and 500 lines of code and scripts that have more than 500 we call large scripts. All scripts were downloaded from open-source projects from github, beside those projects there is a document on the web [tes16] with small scripts that we used as test cases to verify that everything is working as intended and all the errors are found.

On the sections below we demonstrate the tests of the three projects, their analysis and the type of reports that we have already implemented. We choose Captcha (Code available on appendix A) and Cryptobench (Instructions on appendix B) because this scripts are widely used on the web. Captcha script does all verification behind a captcha (we used visualcaptcha project as you can see on figure 5.1)and the Cryptobench is a security benchmark adapted by Google to test performance of their tools some people also use it as examples of cryptography implementations on JavaScript. With this two projects we covered the medium and large size scripts but we still wanted to present an small script so we thought Labyrinth (Code available on appendix C) was an interesting and simple example to present, it randomly generate and draws a labyrinth on a web page as we can see on figure 5.2.

5.2 Module TAJS

From the evaluation of module TAJS, we acquired some results that we will show you in the sections bellow. To demonstrate that, we took some screen shots of the reports system, where we saw how long the analyses took to execute and how many errors were found, as well as we made a comparison between our software and the original TAJS. We denominate the original TAJS as the software that we downloaded from its source.

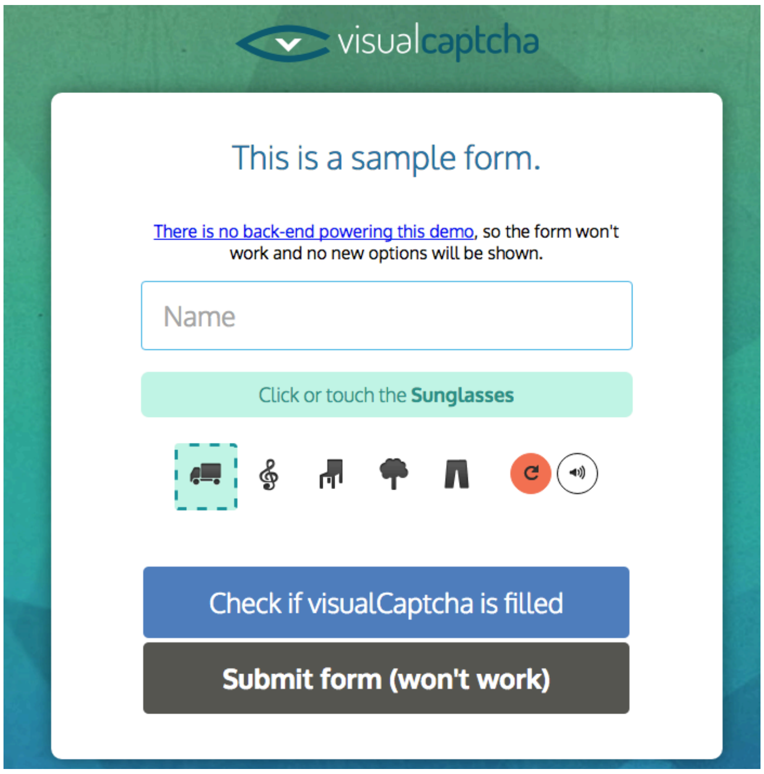

visualcaptcha
This is a sample form.
There is no back-end powering this demo, so the form won't work and no new options will be shown.
Name
Click or touch the Sunglasses
Check if visualCaptcha is filled
Submit form (won't work)

Figure 5.1: Captcha Example

5.2.1 Labyrinth Script

We started by testing TAJS module with the Labyrinth script, a small script that only have seven lines of code, but even small we found three errors as we can see on figure 5.3. This test took 0.458 seconds to run. Comparing our module to the original TAJS we found the same number of errors and the test only took more 0.05 seconds to execute. Every test were run in localhost, so when it will be available on the real world, we must add some latency from the network requests, and an extra time to send the file that we want to test.

5.2.2 Captcha Script

The second script that we executed was the Captcha, a medium size script with 64 lines of code. In this script we have found just one error (figure 5.4), the same as in the original TAJS and test was executed in 0.43 seconds. As seen in our first test we took more 0.03 seconds than the original.

5.2.3 Cryptobench Script

On the Cryptobench script we had amazing results. It is a huge Javascript file, it has 1736 lines of code. Our module could test it in 9.19 seconds and many errors were found, as we can see on figure D.1 (appendix D). Comparing our module to the original TAJS we only took more 0.3 seconds to analyse it and we found the same errors.

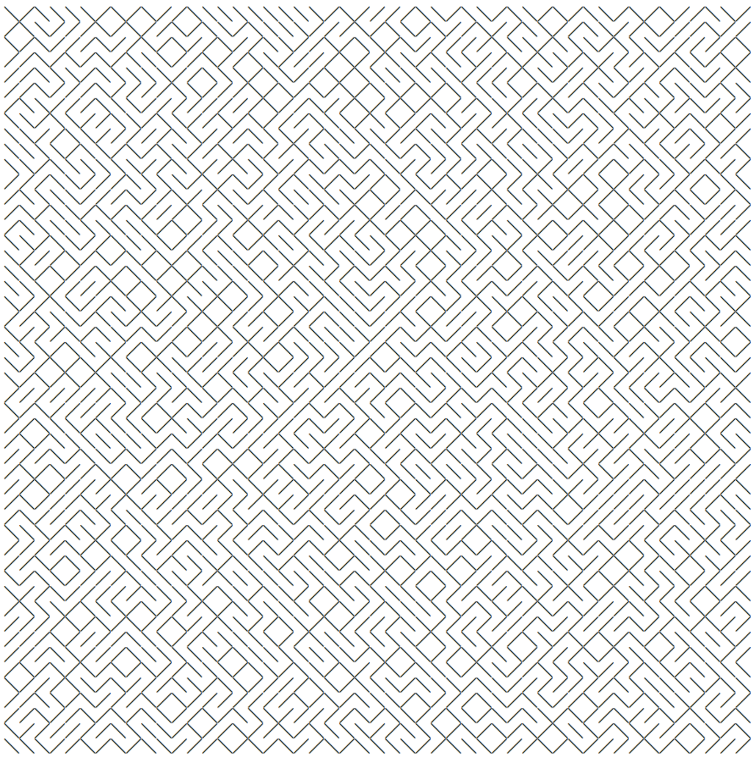


Figure 5.2: Labyrinth Example

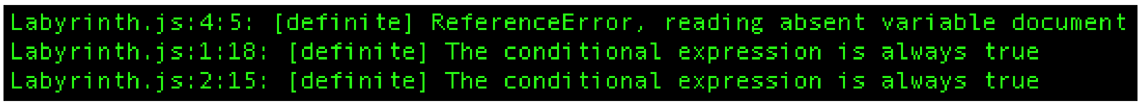

```
Labyrinth.js:4:5: [definite] ReferenceError, reading absent variable document
Labyrinth.js:1:18: [definite] The conditional expression is always true
Labyrinth.js:2:15: [definite] The conditional expression is always true
```

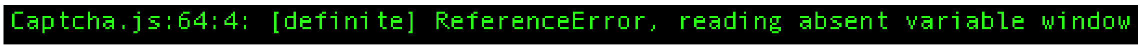
Figure 5.3: Report of TAJS module testing Labyrinth script

5.3 Module JSPrime

As in the segment before, here we will show the evaluation of our JSPrime module on Nibiru framework. On JSPrime we will not compare our module with the original JSPrime because JSPrime is a web page and it has more delays, due to that it's harder to control the environment. So we choose not to compare the time of the executions because our performance is visibly faster, but we checked the errors found and we analysed if they were the same. The reports for JSPrime modules are made in HTML, and are screenshots of it on appendix E and F. However, on Figure 5.5 there is an example of a very small script made to have many problems, in order to we can show the visual of some types of errors that JSPrime module can find. Each type of error has a colour attributed to it, so some errors could have the same background.

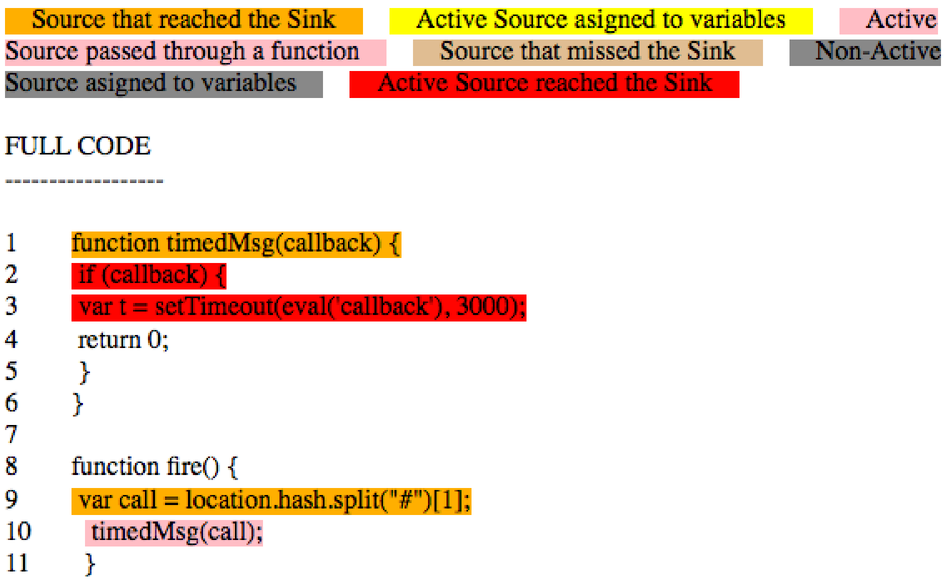
5.3.1 Labyrinth Script

JSPrime module could not find any error on the Labyrinth script. JSPrime does taint analyses and since we don't have possible input from the user, this script is free of taint variables. Our analyses only took 0.1 second to analyse this script and the full report is on appendix E.



```
Captcha.js:64:4: [definite] ReferenceError, reading absent variable window
```

Figure 5.4: Report of TAJS module testing Captcha script



Source that reached the Sink
Active Source asigned to variables
Active
Source passed through a function
Source that missed the Sink
Non-Active
Source asigned to variables
Active Source reached the Sink

FULL CODE

```
------------------

function timedMsg(callback) {
if (callback) {
var t = setTimeout(eval('callback'), 3000);
 return 0;
 }
}

function fire() {
var call = location.hash.split("#")[1];
  timedMsg(call);
 }
```

Figure 5.5: Example of JSPrime report

5.3.2 Captcha Script

On this script we have one input from the user, so there is a taint variable to analyse. JSPrime found the tainted variable and after the analyses it found that the variable missed the sink, it means that variable is tested and it is modified later in the script so the data inside the variable will never do no harm in the software, so it is considered safe variable after the analyse. We analysed the script in 0.15 seconds and the results are more detailed on figure F.1.

5.3.3 Cryptobench Script

Cryptobench script could not be analysed by JSPrime because it is too large. JSPrime taint analyses crashes while we are analysing it because there are too many tainted variables and many modifications to follow. The JSPrime original crashes sooner than on our module, probably because we have more memory/power available so it can stack more taint variables to analyse. On JSPrime original it crashes around 500 lines of code and on our module it crashes around 1100 lines of code, so we saw an improvement on analysing large scripts but not enough.

5.4 Conclusion

We can say that TAJS module on Nibiru works fine. Comparing it to the original TAJS, we only took a little bit more time to run the analyses and its integrity still the same trusted one and now anyone can run analyses on it without the need of his installation. JSPrime module had some improvements that we could measure exactly in terms of time but it works perfectly in small and medium scripts as intended by his developers. In conclusion we saw that in general the type analyses are better to find bugs than the taint analyses.

Chapter 6

Conclusion

OWASP[OWA16] describes “the ten most critical web applications security risks”[OWA13] and shows that our web applications are too insecure often containing many security flaws. To overcome some of this flaws we studied static analysis and realised that this type of analysis can examine JavaScript code, exposing the most critical flaws. This permit us to warn developers, so that they can fix those flaws before the software is released.

This dissertation presented our contribution on the combat of the security risks in web applications by creating a framework called Nibiru. Inside of Nibiru our contribution was the development of this architecture and methods of communication between its modules. It is a framework based on micro-services, where two modules execute security tests by doing static analysis on JavaScript code of the web applications. There is some improvement that can be done on the two security modules presented in this dissertation. The architecture of TAJS module can be simpler and could be optimised to reduce more of the small delays that we have. On other hand in JSPrime there is a problem, it will crash on large scripts (1000 plus lines of JavaScript code), and to fix this problem we need to work on the JSPrime algorithm and completely rewrite it from scratch.

Nibiru showed itself to be a great framework to work with. The work that we have done indicated that Nibiru is already a good framework and works perfectly fine. However, this framework still need much more modules that will generate different types of reports, that will do other type of tests, for example, functional tests or other non-functional tests. For now it only does security, accessibility and usability tests on web applications.

After Nibiru is released to the community, it is hoped that it will grow a lot because it is very useful, simple and easy to develop new modules for it. The software industry will adopt it since they can customise it to their needs and can easily develop modules that we don’t have already. Nibiru is really very versatile and powerful.

6.1 Future Work

There are many improvements that can be made to the Nibiru framework. It has already a prototype that work but many features must be changed, like the number of modules, it needs at least six modules to work properly. Nibiru only has three modules and the three extra modules that must be done before releasing it to the public are the Front-end, DSL Interpretation and Report Generation. The three existing modules plus the modules that need to be done, make Nibiru ready to do most of the non-functional tests on a web application. So as future work in Nibiru it is necessary to develop the three remaining modules and release it to the community so that everyone can use it and make it grow.

The two security modules of Nibiru can find most of the flaws listed by OWASP [OWA13], to cover them all we needed to do JSFlow module. JSFlow will help with other type of analysis on web applications and with three modules (JSPrime, TAJS and JSFlow) executing tests is reliable to say that an application that pass all the tests is heavily tested.

The future work on JSPrime is around its capacity to analyse the large size scripts, through some optimisations on its algorithm.

Bibliography

[Boj15] Valentin Bojinov. *RESTful Web API Design with Node.js*. Packt Publishing - ebooks Account, 2015. Available from: `http://gen.lib.rus.ec/book/index.php?md5=7906fa2a22be0dfbbb31441461db9352`. 11

[Bur03] Ilene Burnstein. *Practical Software Testing*. Springer, 2003. 6

[CPSF16] Joel Carvalho, Nuno Pereira, Simão Sousa, and João Fernandes. Automated analysis of non-functional requirements for web applications, 2016. Available from: `http://www.aistic.org/cisti2016/oc16/modules/request.php?module=oc_program&action=summary.php&id=127`. xi, 24, 25

[flo16] Flow webpage, 2016. Available from: `https://flowtype.org/`. 18

[Fow14] Martin Fowler. Microservices a definition of this new architectural term, 2014. Available from: `http://martinfowler.com/articles/microservices.html`. 13

[gcc16] Gcc, the gnu compiler collection, 2016. Available from: `https://gcc.gnu.org/`. 8

[icl14a] Apple icloud security exploit is a concern, experts say, 2014. Available from: `http://www.bbc.com/news/technology-29045789`. iii, v, vii, 1

[icl14b] Hack leaks hundreds of nude celebrity photos, 2014. Available from: `http://www.theverge.com/2014/9/1/6092089/nude-celebrity-hack`. iii, v, vii, 1

[Inf12] Information-flow security for a core of javascript, 2012. Available from: `http://www.cse.chalmers.se/~andrei/jsflow-csf12.pdf`. 17

[inf16] Infer webpage, 2016. Available from: `http://fbinfer.com/`. 18

[iso86] Software quality terminology, 1986. Available from: `http://www.issco.unige.ch/en/research/projects/ewg95/node69.html`. 6

[Joe16] Joel carvalho webpage, 2016. Available from: `http://joelcarvalho.pt/`. 24

[jsf14] Jsflow: Tracking information flow in javascript and its apis, 2014. Available from: `http://www.cse.chalmers.se/~andrei/sac14.pdf`. 17

[JSP16a] Github page to jsprime module source code, 2016. Available from: `https://github.com/kroglice/NibiruJSPrimeModule`. 26

[jsp16b] Jsprime, 2016. Available from: `http://dpnishant.github.io/jsprime/`. 17, 24

[Mar09] Robert C. Martin's. The single responsibility principle, 2009. Available from: http://programmer.97things.oreilly.com/wiki/index.php/\The_Single_Responsibility_Principle. 13

[New15] Sam Newman. *Building Microservices*. O'Reilly Media, Inc., 1005 Graenstein Highway North, Sebastopol, CA 95472, 2015. 12

[Nie05] Flemming Nielson. *Principles of Program Analysis*. Springer, 2005. 15, 16

[OWA13] OWASP. Owasp top 10 - the ten most critical web application security risks, 2013. Available from: http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202013.pdf. 8, 35, 36

[OWA16] OWASP. Owasp main web page, 2016. Available from: https://www.owasp.org/index.php/Main_Page. 8, 35

[Pat05] Ron Patton. *Software Testing*. Sams, 2005. 3, 6

[RES08] Restful web services: The basics, 2008. Available from: http://www.gregbulla.com/TechStuff/Docs/ws-restful-pdf.pdf. 11, 12

[rhi16] Rhino, 2016. Available from: https://github.com/mozilla/rhino. 18

[sca16] Scanjs github webpage, 2016. Available from: https://github.com/mozilla/scanjs. 18

[Sin11] Yogesh Singh. *Software Testing*. Cambridge, 2011. 3, 4, 6, 7

[Soc99] The Internet Society. Hypertext transfer protocol – http/1.1, 1999. Available from: https://tools.ietf.org/html/rfc2616. 11

[TAJ16a] Github page to tajs module source code, 2016. Available from: https://github.com/kroglice/Nibiru-TAJS-module. 25

[taj16b] Type analyzer for javascript, 2016. Available from: http://www.brics.dk/TAJS/. 18, 24

[tes16] Document with javascript code, 2016. Available from: https://docs.google.com/document/d/17J2h43WbPX3sNTIjxr4GhzEGxKy6hvQJqedd3UQ11mY/edit. 29

[WHA13] What is security testing?, 2013. Available from: https://www.securitycompass.com/media/pdf/article-what-is-security-testing.pdf. 7, 8

Appendix A

Captcha

```
(function (window, visualCaptcha) {
    var captcha = visualCaptcha('sample-captcha', {
        imgPath: 'img/',
        captcha: {
            numberOfImages: 5,
            callbacks: {
                loaded: function (captcha) {
                    // Avoid adding the hashtag to the URL when clicking/selecting visualCaptcha options
                    var anchorOptions = document.getElementById('sample-captcha').getElementsByTagName('a');
                    var anchorList = Array.prototype.slice.call(anchorOptions);// .getElementsByTagName does not return an actual array
                    anchorList.forEach(function (anchorItem) {
                        _bindClick(anchorItem, function (event) {
                            event.preventDefault();
                        });
                    });
                }
            }
        }
    });

    var statusElement = document.getElementById('status-message'),
        queryString = window.location.search;
    // Show success/error messages
    if (queryString.indexOf('status=noCaptcha') !== -1) {
        statusElement.innerHTML = '<div class="status"> <div class="icon-no"></div> <p>visualCaptcha was not started!</p> </div>' + statusElement.innerHTML;
    } else if (queryString.indexOf('status=validImage') !== -1) {
     statusElement.innerHTML = '<div class="status valid"> <div class="icon-yes"></div> <p>Image was valid!</p> </div>' + statusElement.innerHTML;
 } else if (queryString.indexOf('status=failedImage') !== -1) {
     statusElement.innerHTML = '<div class="status"> <div class="icon-no"></div> <p>Image was NOT valid!</p> </div>' + statusElement.innerHTML;
 } else if (queryString.indexOf('status=validAudio') !== -1) {
     statusElement.innerHTML = '<div class="status valid"> <div class="icon-yes"></div> <p>Accessibility answer was valid!</p> </div>' + statusElement.innerHTML;
 } else if (queryString.indexOf('status=failedAudio') !== -1) {
     statusElement.innerHTML = '<div class="status"> <div class="icon-no"></div> <p>Accessibility answer was NOT valid!</p> </div>' + statusElement.innerHTML;
 } else if (queryString.indexOf('status=failedPost') !== -1) {
     statusElement.innerHTML = '<div class="status"> <div class="icon-no"></div> <p>No visualCaptcha answer was given!</p> </div>' + statusElement.innerHTML;
 }


 // Binds an element to callback on click
 // @param element object like document.getElementById() (has to be a single element)
 // @param callback function to run when the element is clicked
 var _bindClick = function (element, callback) {
     if (element.addEventListener) {
         element.addEventListener('click', callback, false);
     } else {
         element.attachEvent('onclick', callback);
     }
 };

 // Show an alert saying if visualCaptcha is filled or not
 var _sayIsVisualCaptchaFilled = function (event) {
     event.preventDefault();
```

```

        if (captcha.getCaptchaData().valid) {
            window.alert('visualCaptcha is filled!');
        } else {
            window.alert('visualCaptcha is NOT filled!');
        }
    };

    // Bind that function to the appropriate link
    var isFilledElement = document.getElementById('check-is-
   filled');
    _bindClick(isFilledElement, _sayIsVisualCaptchaFilled);
} (window, visualCaptcha));
```

Listing A.1: Captcha Script

Appendix B

Cryptobench

Source code available online on **https://people.mozilla.org/ sfink/duh/code/crypto.txt** web page due to is large size.

Appendix C

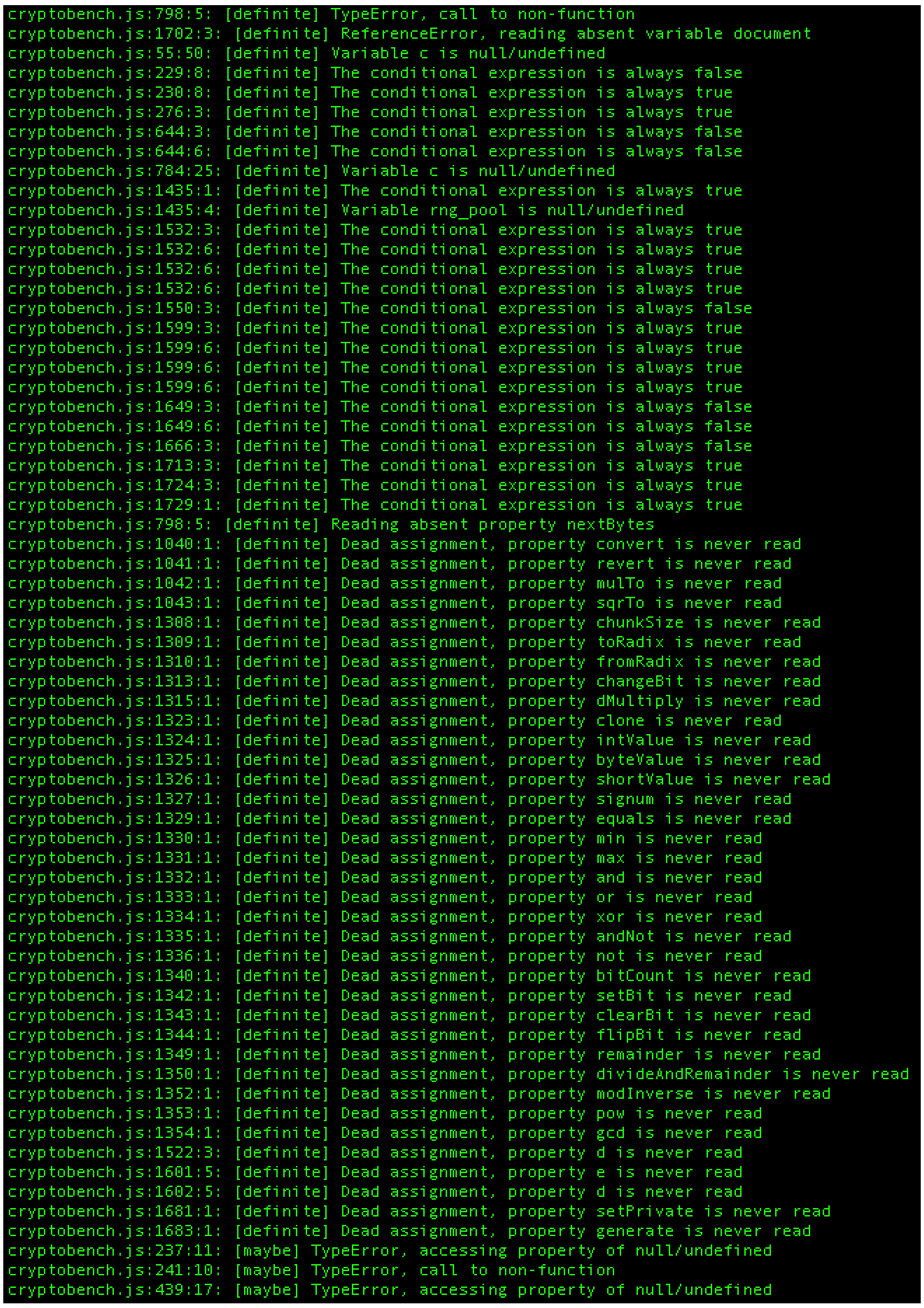
Labyrinth

```
<p style="line-height: 18px; font-size: 18px;  font-family:
   times;">
<script>
for (var line=1; line<50; line++) {
  for(var i=1;i<50;i++) {
    var s = (Math.floor((Math.random()*2)%2)) ? "&#9585" : "
   &#9586";
    document.write(s);
  }
  document.writeln("<br>");
}
</script>
</p>
```

Listing C.1: Labyrinth Script

Appendix D

TAJS report for Cryptobench

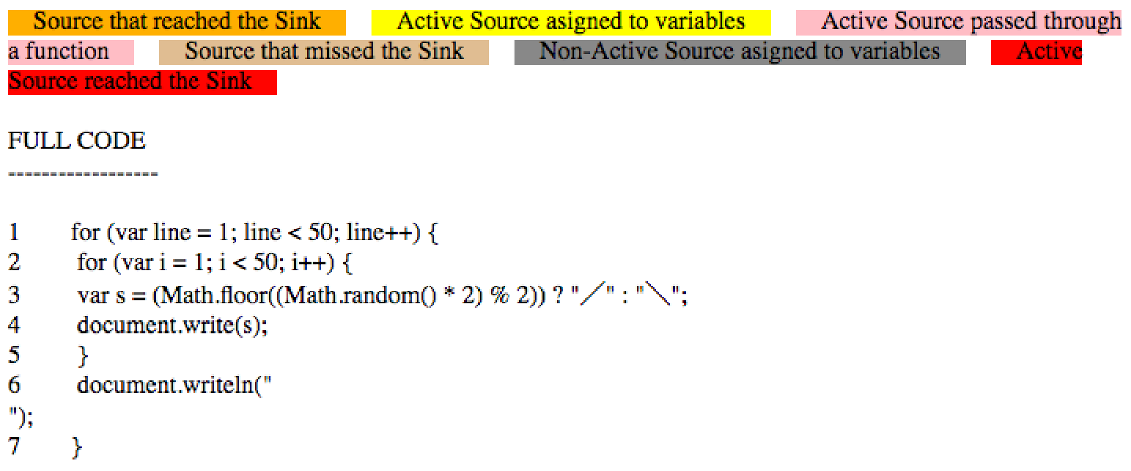


```
cryptobench.js:798:5: [definite] TypeError, call to non-function
cryptobench.js:1702:3: [definite] ReferenceError, reading absent variable document
cryptobench.js:55:50: [definite] Variable c is null/undefined
cryptobench.js:229:8: [definite] The conditional expression is always false
cryptobench.js:230:8: [definite] The conditional expression is always true
cryptobench.js:276:3: [definite] The conditional expression is always true
cryptobench.js:644:3: [definite] The conditional expression is always false
cryptobench.js:644:6: [definite] The conditional expression is always false
cryptobench.js:784:25: [definite] Variable c is null/undefined
cryptobench.js:1435:1: [definite] The conditional expression is always true
cryptobench.js:1435:4: [definite] Variable rng_pool is null/undefined
cryptobench.js:1532:3: [definite] The conditional expression is always true
cryptobench.js:1532:6: [definite] The conditional expression is always true
cryptobench.js:1532:6: [definite] The conditional expression is always true
cryptobench.js:1532:6: [definite] The conditional expression is always true
cryptobench.js:1550:3: [definite] The conditional expression is always false
cryptobench.js:1599:3: [definite] The conditional expression is always true
cryptobench.js:1599:6: [definite] The conditional expression is always true
cryptobench.js:1599:6: [definite] The conditional expression is always true
cryptobench.js:1599:6: [definite] The conditional expression is always true
cryptobench.js:1649:3: [definite] The conditional expression is always false
cryptobench.js:1649:6: [definite] The conditional expression is always false
cryptobench.js:1666:3: [definite] The conditional expression is always false
cryptobench.js:1713:3: [definite] The conditional expression is always true
cryptobench.js:1724:3: [definite] The conditional expression is always true
cryptobench.js:1729:1: [definite] The conditional expression is always true
cryptobench.js:798:5: [definite] Reading absent property nextBytes
cryptobench.js:1040:1: [definite] Dead assignment, property convert is never read
cryptobench.js:1041:1: [definite] Dead assignment, property revert is never read
cryptobench.js:1042:1: [definite] Dead assignment, property mulTo is never read
cryptobench.js:1043:1: [definite] Dead assignment, property sqrTo is never read
cryptobench.js:1308:1: [definite] Dead assignment, property chunkSize is never read
cryptobench.js:1309:1: [definite] Dead assignment, property toRadix is never read
cryptobench.js:1310:1: [definite] Dead assignment, property fromRadix is never read
cryptobench.js:1313:1: [definite] Dead assignment, property changeBit is never read
cryptobench.js:1315:1: [definite] Dead assignment, property dMultiply is never read
cryptobench.js:1323:1: [definite] Dead assignment, property clone is never read
cryptobench.js:1324:1: [definite] Dead assignment, property intValue is never read
cryptobench.js:1325:1: [definite] Dead assignment, property byteValue is never read
cryptobench.js:1326:1: [definite] Dead assignment, property shortValue is never read
cryptobench.js:1327:1: [definite] Dead assignment, property signum is never read
cryptobench.js:1329:1: [definite] Dead assignment, property equals is never read
cryptobench.js:1330:1: [definite] Dead assignment, property min is never read
cryptobench.js:1331:1: [definite] Dead assignment, property max is never read
cryptobench.js:1332:1: [definite] Dead assignment, property and is never read
cryptobench.js:1333:1: [definite] Dead assignment, property or is never read
cryptobench.js:1334:1: [definite] Dead assignment, property xor is never read
cryptobench.js:1335:1: [definite] Dead assignment, property andNot is never read
cryptobench.js:1336:1: [definite] Dead assignment, property not is never read
cryptobench.js:1340:1: [definite] Dead assignment, property bitCount is never read
cryptobench.js:1342:1: [definite] Dead assignment, property setBit is never read
cryptobench.js:1343:1: [definite] Dead assignment, property clearBit is never read
cryptobench.js:1344:1: [definite] Dead assignment, property flipBit is never read
cryptobench.js:1349:1: [definite] Dead assignment, property remainder is never read
cryptobench.js:1350:1: [definite] Dead assignment, property divideAndRemainder is never read
cryptobench.js:1352:1: [definite] Dead assignment, property modInverse is never read
cryptobench.js:1353:1: [definite] Dead assignment, property pow is never read
cryptobench.js:1354:1: [definite] Dead assignment, property gcd is never read
cryptobench.js:1522:3: [definite] Dead assignment, property d is never read
cryptobench.js:1601:5: [definite] Dead assignment, property e is never read
cryptobench.js:1602:5: [definite] Dead assignment, property d is never read
cryptobench.js:1681:1: [definite] Dead assignment, property setPrivate is never read
cryptobench.js:1683:1: [definite] Dead assignment, property generate is never read
cryptobench.js:237:11: [maybe] TypeError, accessing property of null/undefined
cryptobench.js:241:10: [maybe] TypeError, call to non-function
cryptobench.js:439:17: [maybe] TypeError, accessing property of null/undefined
```

Figure D.1: Report of TAJS module testing Cryptobench script

Appendix E

JSPrime report for Labyrinth



Source that reached the Sink Active Source asigned to variables Active Source passed through a function Source that missed the Sink Non-Active Source asigned to variables Active Source reached the Sink

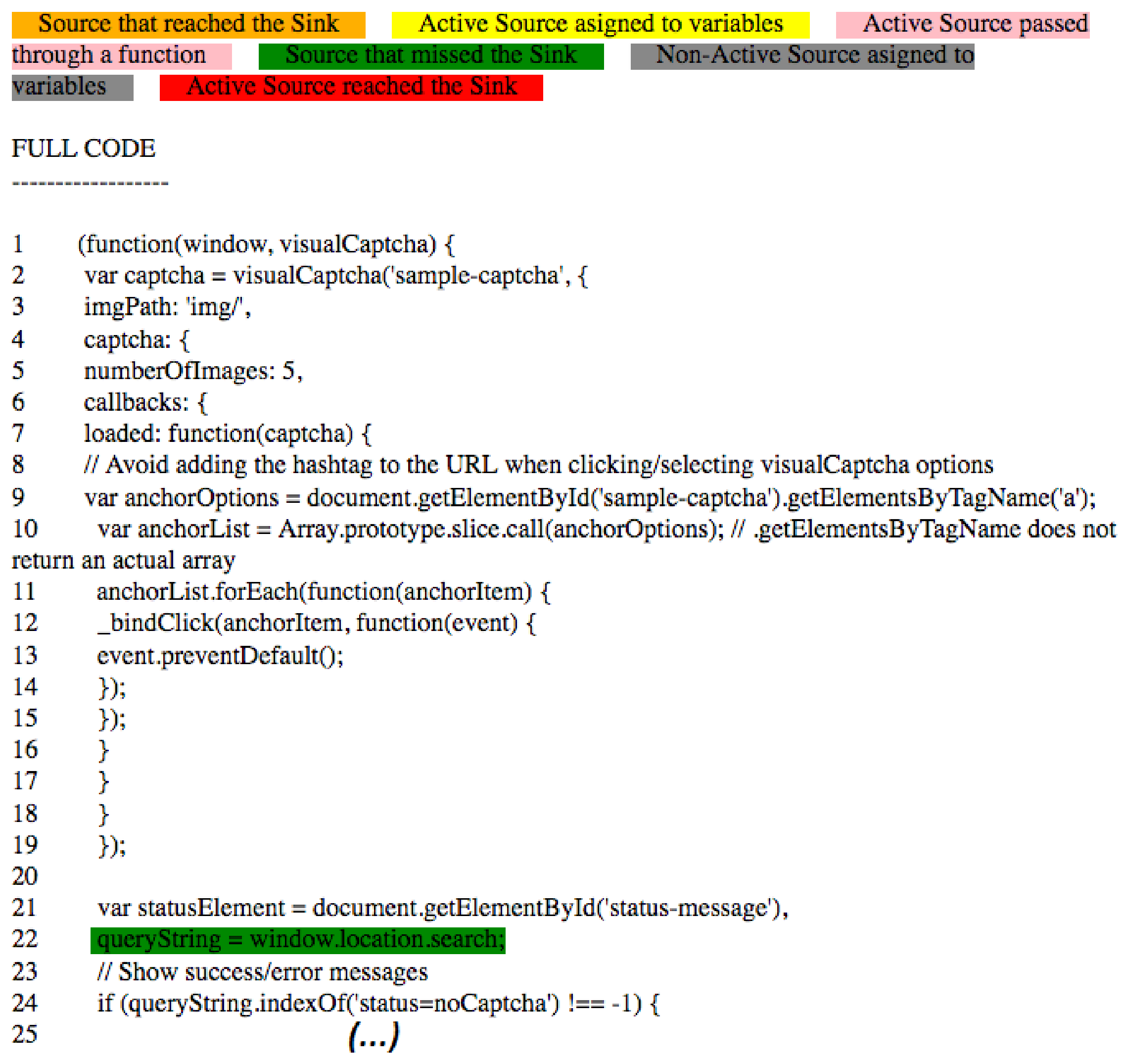
FULL CODE

```
for (var line = 1; line < 50; line++) {
 for (var i = 1; i < 50; i++) {
 var s = (Math.floor((Math.random() * 2) % 2)) ? "╱" : "╲";
 document.write(s);
 }
 document.writeln("
");
}
```

Figure E.1: JSPrime report for Labyrinth

Appendix F

JSPrime report for Captcha



Source that reached the Sink Active Source asigned to variables Active Source passed through a function Source that missed the Sink Non-Active Source asigned to variables Active Source reached the Sink

FULL CODE

```
(function(window, visualCaptcha) {
 var captcha = visualCaptcha('sample-captcha', {
 imgPath: 'img/',
 captcha: {
 numberOfImages: 5,
 callbacks: {
 loaded: function(captcha) {
 // Avoid adding the hashtag to the URL when clicking/selecting visualCaptcha options
 var anchorOptions = document.getElementById('sample-captcha').getElementsByTagName('a');
  var anchorList = Array.prototype.slice.call(anchorOptions); // .getElementsByTagName does not return an actual array
  anchorList.forEach(function(anchorItem) {
  _bindClick(anchorItem, function(event) {
  event.preventDefault();
  });
  });
  }
  }
  }
  });

  var statusElement = document.getElementById('status-message'),
  queryString = window.location.search;
  // Show success/error messages
  if (queryString.indexOf('status=noCaptcha') !== -1) {
                       (...)
```

Figure F.1: JSPrime report for Captcha