Dissertation

Masters in Informatics Engineering

An Integrated Tool to Detect Vulnerabilities in Service-Based Infrastructures

Rafael Simões Ventura
ventura@dei.uc.pt

Advisor:
Prof. Marco Vieira
Co-advisor:
Dr. Nuno Antunes

02-09-2014
Department of Informatics Engineering
Faculty of Science and Technology
UNIVERSITY OF COIMBRA

Abstract

A Service-based infrastructure is a composition of software pieces that provides functionalities that supports organizational large operations. Although they can be composed by any kind of services, web services are often the preferred technology used, due to their characteristics. This kind of infrastructures can be frequently found in online stores, educational and banking systems, making the web services used business-critical components. However, studies show that many of these services are deployed disregarding security concerns, which results in a huge amount of vulnerable web services in production.

The cost of manual code inspection and lack of security expertise of programmers brought the need to use automatic vulnerability scanning tools, yet studies and reports show that the effectiveness of these scanners is insufficient. In fact, experiments from previous work show that existing scanners present a low detection rate and a high number of false positives. To address this problem we need new and improved tools.

This works presents a new integrated tool implemented in a modular fashion where the components of vulnerability scanning techniques can be easily integrated without the need of implement new tools for each new technique. With this tool it is possible not only to implement the known techniques but it also enables the evolvement of the same. Among with this tool were built the necessary modules to complete three different techniques. A benchmark was applied to the tool and the results show viability of the modular approach of the integrated tool.

Keywords

Automated Tools, Black-box Testing, Extensible Tools, Gray-box Testing, Modular Development, Security, Service Oriented Architecture, Software Testing, Software Vulnerabilities, Vulnerability Detection, Web Services

Acknowledgements

I would like to start to thank to my son for the funny moments he is always propose to me in the few moments we had to be together during this work. Also for the capacity he has to make me laugh when the only think I wanted to do was to cry.

Second I wan to thank to Cátia, for her support on during all this time and also for doing a great job taking care of our son in my absence.

A special thank to my parents and my brother that always provided me all the support and resources to whatever I need.

I need also to thank the help of all my friends.

Last but not the least I want to thank to my advisors, Professor Marco Vieira and Professor Nuno Antunes, I want thank them for their help, guidance and mainly their patience and friendship, with them this work wouldn't be possible.

Table of Contents

List of Figures

Chapter 1 **Introduction**

Enterprise world is in constant change [1] and in order to enterprises achieve their goals they need to define processes and workflows to improve productivity. If this is true for human resource management, it also true for information technology. Additionally, enterprise business' needs are constantly changing and the information technologies should adapt accordingly. Some year ago appeared a new architecture style known by **Service-Oriented Architectures** (SOA) and its acceptance has escalated until now due to its base principals. In this kind of architectures, every business rule, policy or activity should be independent, decoupled and context aware, thus when enterprises need to define new business processes they just need to orchestrate all of the available activities in order to provide some context managing and relating existing activities in order to accomplish a process [1]. Due to its characteristics, Service-based Infrastructures are very important since they meet all of the requirements of this architecture style, representing the main support to Service-Oriented Architectures.

A **service-based infrastructure** consists of an aggregation of software components that interact with each other in order to support business-critical processes of organizations. These software components, also referred as services, are well defined, self-contained, platform and context independent modules that provide a specific functionality [2]. These services can be implemented using different technologies, however web services are usually the elected choice of implementation.

Although these services are commonly required to be secure the truth is that often they are developed and deployed disregarding security aspects, probably due to the fact that programmers focus on functionality and design instead of security [3]. This lack of attention regarding security, together with the fact that services are very exposed, make them a preferred target of attackers. This also means that any security breach that they have will eventually be found and exploited. According with OWASP top ten 2013 [4], injection vulnerabilities are nowadays the most critical for web applications. Although web services and web application are different types of applications, they share many of the concerns regarding security. In fact, the results published in [5] show that there are a large amount of web services over the internet that are vulnerable to injection attacks.

An injection attack can be observed when someone provides a malicious input to an application with the intent of changing the structure of the backend commands. Successful injection attacks allows attackers to execute unwanted commands that can be used to create, delete or modify data, thus compromising the application and even its underlying systems [6]. These attacks are particularly dangerous because they can easily bypass firewalls, Intrusion Detecting Systems and other network communication countermeasures, as they are conducted through the same ports that are used for regular use of the application [7]. Even web application firewalls are unable to defend these attacks, as that requires a deep understanding of the business context[8]. Examples of vulnerabilities in this group are SQL Injection where the attacker provides SQL code as input in order to change the SQL command logic[9] and XPath injection, that similarly occurs when some XML is provided as input in order to modify a query to an XML database [10].

The best programming practices should be applied in order to avoid the weaknesses and flaws in the code that may cause software to be vulnerable. Additionally, code reviews and inspections are essential to prevent vulnerabilities. In fact, during these processes it is possible to uncover a big percentage of the bugs that can lead to security breaches. However, it is also a fact that the process of reviewing and inspecting code represents a massive, tedious and, above all, expensive task. This way, automatic vulnerability detection tools became very important on helping programmers to develop more secure services thus making service-based infrastructures also more secure.

There are available several tools that automate security tests and scanners that detect vulnerabilities; these tools can be categorized by the approaches that they follow:

- **White-box analyses** consist of analyzing the service source code without executing it, searching for the patterns of security flaws. It can be done manually or automatically using source code analyzers such as: FindBugs [11], IntelliJ IDEA Static Code Analysis [12] , FlawFinder[13].
- **Black-box testing** also known as penetration testing consists in the execution analyses of the application while it's under attack, basically fuzzing techniques are used to test applications, in fact applications are tested being attacked while vulnerability detection is performed based on the responses analyses WSDigger [14], WSFuzzer [15] are examples of open source tools, also, commercial tools such as: HP Webinpect[16], IBM Rational AppScan [17], Acunetix Web Vulnerability Scanner[18].
- **Gray-box testing** that combines characteristics of both techniques with the goal overcoming their limitations while keeping their advantages. There is an approach named as AMNESIA proposed in [19]. Acunetix also developed a commercial tool using this technique Acunetix WVS AcuSensor [20].

From all these techniques, black-box is probably the most used, mainly because in service-based infrastructures the target service to test may be not owned by the test carrier, thus white-box analyses is not helpful since it needs access to the source code. Another advantage of penetration testing over code analyses techniques is the fact that provides a dynamic perspective of the execution of the service during testing what enables a better representation of a real-world environment. This way, in this work we are particularly interested in testing based techniques.

Previous studies evaluated some of the most popular vulnerability detection tools and it was shown that the effectiveness of these tools could vary a lot, with the unsatisfactory results being the common pattern. In [21] some of this tools have been benchmarked and the results showed that most of these tools have at least one of the following two drawbacks: low detection rate and high false positives rate.

Such results opened the possibility to propose some new and improved approaches for detecting injection vulnerabilities in web services. In [22] the authors proposed an improved penetration testing approach to detect SQL Injection vulnerabilities (IPT-WS). The approach uses representative workloads, implements effective attackloads, and applies well-defined rules to analyze the web services responses, to improve detection coverage while reducing false positives. To overcome the limitations of penetration testing, the authors proposed an approach based on attack signatures and interface monitoring to detect injection vulnerabilities (Sign-WS)[23]. The approach introduces special tokens inside the injection attacks (signatures) and then monitors the interfaces of the service under testing looking for these tokens, detecting more vulnerabilities while avoiding false positives. Finally, in [24] it is proposed runtime anomaly detection approach to detect injection vulnerabilities in web services (RAD-WS), that takes advantage of information about the internal behavior of the application can be used to achieve maximum effectiveness, in the scenario where it is possible to access the source code (or p-code). The approach exercises the service for profiling its regular internal behavior (learning phase) and then attacks the service (attacking phase), reporting vulnerability when some deviation is detected.

1.1. **Extending Vulnerability Detection Tools**

Although these techniques presented promising results, there is room for improvement. The prototypes for these tools were implemented in an *ad-hoc* fashion, without following a standardized architecture or a standard procedure to develop vulnerability detection tools, raising several problems:

1. The effort required to make changes to the existing prototypes is huge because a small change in the tool may lead to modifications in several parts of the code. This makes it very hard for the tools to be extended;
2. The tools are very hard to maintain by other developers that were not involved in their implementation. This difficulties the reuse of the tools to extend with new and improved algorithms and techniques;
3. Most of the techniques share some functionality, what means that large chunks of source code are replicated in all of the three existing tools. However, when one of the tools is improved in these functionalities, the other tools do not benefit;
4. It is necessary to run different tools to use different techniques. This makes their use harder in this kind of environment, where several services are necessary to be tested and many times with different techniques;

In order to overcome these limitations it was necessary to develop a new tool with the ability of integrating new approaches and which provides an easy way to evolve and improve them. Such tool needed to allow both manual input and automatic discovering resources of the infrastructure in order to test the infrastructure using the most adequate approach to each service.

This work presents a new tool that is able to integrate several techniques with the composition of eternal modules working together, thanks to its new architecture. The tool is also capable of monitoring the infrastructure to automatically discover new resources and relationship between them thus, dealing with dynamicity of service-based infrastructures.

To empower the tool, it was implemented an API that contains all the specification needed to implement new external modules. Thus, the unique dependency of a specific implementation of an new module is this API that is provided as a library. Also the tool already includes specific implementation of several modules that working together can build different vulnerability detection techniques. This can be achieved by orchestrating the external modules with a core module that is composed by several internal components. Those interact with each other plus with the external modules that have also been implemented. Some of the internal components generate information to the external modules and the external modules produces and provide the necessary data to enable vulnerability scanning. Modules such as workload generation, attackload generation, monitoring and discovering, and detection are external to the tool and its implementation is transparent to the core module, their communication in made through the implemented API that contains the interfaces and the information that needs to be shared between internal and external components, this keeps any required modification in one the modules independent and way more maintainable. In addition, new instances of the modules can be easily plugged to the core module without modifying any other part of the core.

We have applied a benchmark to our tool that has been already applied to the exitsing tools. The results of benchmarks shows that is possible to implement this three approaches using our modular tool, and shows also that is possible to have all this tools integrated in the same tools. Thus we believe that our tool turns the development of novel techniques and evolve the existing one much easier then before and at the same time encourage people to collaborate with single modules implementation that can be used by the community to advance in the research the vulnerability detection techniques.

1.2. **Contribution**

The goal of this dissertation work was to develop an application that could integrate different vulnerability detection techniques in the same application, also we wanted to perform scan to a set of services dynamically changing the technique used depending on the access the test carrier has to it. This way, the main contributions of this work are:

- A modular design for the vulnerability detection techniques, enabling the composition of techniques using external modules
- Implementation of a core module to manage external modules and orchestrate the interaction and communication between external modules.
- Implementation of an API that allows developers to easily extend the application without the need to had knowledge about the entire tool.
- Implementation of the necessary modules to emulate three different techniques: IPT-WS, Sign-WS and RAD-WS. These modules include workload generators, attackload generators, monitoring, detecting and also probes that will be deployed in the target system and gather information to the monitoring module.

1.3. **Document structure**

The content of this document is divided in Chapters; the organization of the content is as follows: The Chapter 2 presents the review on the background and related work, the Chapter 3 shows the design decisions taken in order to complete the system requirements, Chapter 4 gives some details about the implementation of the integrated too as well as API and external modules, Chapter 5 have an evaluation of the work, Chapter 6 present a developer and a user guide of the tool, Finally Chapter 7 concludes this document.

Chapter 2
Background and Related Work

In the context of this work it was necessary to understand the concepts involving the security of web service's code. Regarding this, it was performed an extensive review of the state-of-the-art on web services, vulnerabilities, vulnerability detection techniques and existing tools. During this research, it was identified which are some common components that a vulnerability scanner shall contain, namely workload generator, attackload generator, monitoring and discovering, and detecting. A review of the state-of-the-art in these topics was performed as well.

2.1. **Services and Services Infrastructures**

A web service is a piece of business logic available over a network that provides a simple interface between a provider and a consumer. In SOAP web services [25] this interface is in shape of operations, which are, in practice, methods that receives parameter and return an output. These interfaces can be used over an Internet protocols like HTTP or HTTPS. The communication between a provider and a consumer consists in exchanging Simple Object Access Protocol (SOAP) messages. SOAP is XML-based (eXtendable Markup Langue) format, hence allowing data and complex data representation as XML elements assuring that the message can parsed by any platform, thus assuring the interoperability of this kind of services [25]. The process of a communication starts with the client building a SOAP message and sending it to the server, once the message has been processed, the server builds a SOAP message with the response and sends it to the client.

Web services are described by WSDL files [26], which consist of a XML document file that contains information about operations by the web services and respective parameters and return values provided (e.g. data types, parameter domains, etc.). This way the consumer of the service can easily and in an automated fashion understand the operations available and how to construct valid SOAP requests to the service. At the same type consumer also know the response type for each operation, facilitating its processing. In order to be advertised and discovered web services can be registered in a UDDI (Universal Description, Discovering and integration)[26], in the same way UDDI permits others to search for web services. UDDI is a platform-independent framework for describe [26] discover and integrate business services, actually it can be seen as a directory with information about web services and the respective WSDL files, the communication with UDDI is made (again) via SOAP. Obviously this kind of services need an infrastructure to operate, the infrastructure is usually composed by operating systems, application servers and some times external resources (DBMSs, XML databases, etc.) and even by other web services as represented in **Figure 1**.

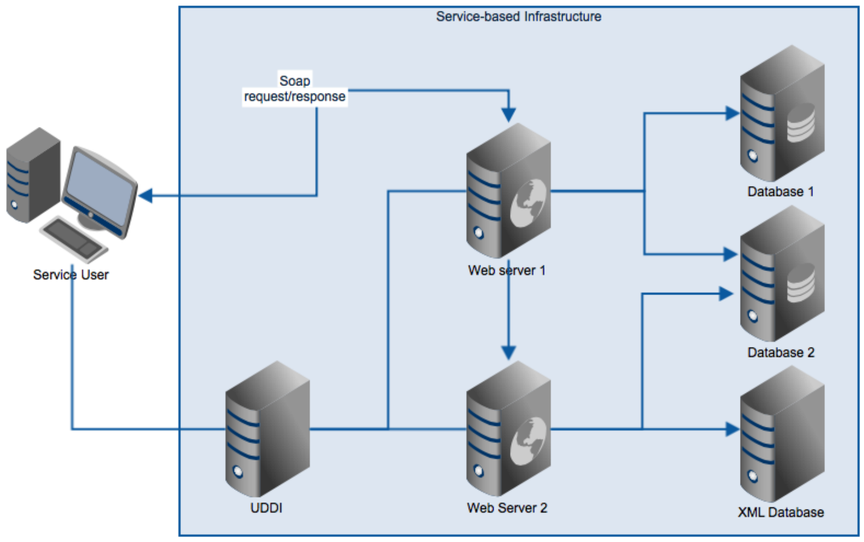


Figure 1 – Typical Service-based Infrastructure

Another well-known class of web services are RESTful services (Representational State Tranter) [27] services that started to be widely used being, nowadays, increasingly adopted. Their advantages are simplicity, and, different of SOAP web services, here the service exposes its operations as resources. This means that for each operation it's provided an URI, that can be requested using native HTTP methods (GET, PUT, POST, DELETE) with the required parameter, the response its retrieved in XML or JSON format, explicitly requested as one of the parameters. The big difference between Restful services and Web Services is that they even if they use an HTTP to exchange messages they do it in different ways, in Restful the requests goes straight away in HTTP body that will response with an XML or JSON [27] also in the body of HTTP response, instead of that Web Services uses HTTP to transfer XML documents with the requests and responses. Similarly to Web Services, RESTful services may have a description file that provides all the information necessary to interact with RESTful resources WADL (Web Application Description Language)[27] however, in the case of RESTful services the use of this files is not mandatory, actually due to the simplicity of RESTful interfaces are defined this description file is barely used.

Due to the characteristics of web services and service-based infrastructures, we can identify three distinct testing scenarios representing the information and level of access that the user can have to each service:

- **Services under control:** a service is under control and also the resources that it uses are known. This service uses only services and resources owned by a provider to complete its business functionality. It is possible to use all kinds of vulnerability detection techniques, including the ones that require access to the source code (e.g. static analysis).
- **Services partially under control:** a service is under control but some of the resources that it uses are not. In practice, this service requires services and/or resources that are not owned by a provider to complete its business functionality. In this case, it may not be possible to access the source code (e.g. in the case of legacy systems or systems based in off-the-shelf components). However, all the interfaces between the service and the external environment are known, which

allows one to obtain relevant information about input domains and about the use of external resources. This allows using techniques that take advantage of such information (e.g. techniques based on interface monitoring).

- **Services within reach:** a service is within reach but not under control. This means that a provider is able to invoke this service, but not to control it (e.g. has no access or detailed information about the service). As it is not possible to access the internals of the service, only black-box testing techniques can be used. This scenario also represents the typical point-of-view of the consumer.

2.2. **Services' Threats and Vulnerabilities**

The intrinsic characteristics of web services make them one of the preferred methods of users to access data over the network. A single system may use or provide several web services which represents a large number of entry points to access data, and at the same time it increases the entry points to attacks. A vulnerability is a flaw or weakness in a system design, implementation, or operation and management that can be exploited to violate the systems security policy [4]. In that sense there are a lot of vulnerabilities that could be found on web services.

As web services are so widely spread and exposed, any existing vulnerability will most likely be uncovered and exploited by attackers. According with studies published in May 2013 by Whitehatsec, 86% of the web sites present at least, one serious vulnerability [28]. This study also demonstrates that the most prevalent web application vulnerabilities are: Information Leakage, cross-site scripting, content spoofing, etc. Although SQL Injection came in 14th place on the top prevalent vulnerability with 7% of the web sites showing at least one serious vulnerability of this kind we should consider that some of the vulnerabilities are not applicable or common on web services as they are to web applications, as an example XSS (cross site scripting) is not considered critical because typically a web service is not attached to a web site. At the same time we also consider that Information Leakage can be one of the consequences of SQL Injection vulnerabilities, so we assume that this number can by much higher then the one presented on Whitehatsec study. Another main consideration is the criticality of the vulnerability, OWASP Top 10 2013 consider Injection vulnerabilities the most critical vulnerabilities that may appear on web applications [4].

Web services have similar problems, as shown by the field study presented in [5]. The authors tested 300 public web services using four well-known vulnerability scanners and observed that in more than 8% of the services vulnerabilities were found. Additionally, the most frequent vulnerabilities were injection vulnerabilities (SQL Injection and XPath Injection), by far the most reported vulnerabilities by the four scanners. An injection attack occurs when users maliciously inputs code in order to modify the application behavior with the intent of be granted privileged access that allows him to have control of the system or access to sensitive information or modify the integrity of data [4].

2.3. **Vulnerability Detection Approaches**

To minimize security issues, security must be present during the entire software product development lifecycle [3]. However, one of the most important parts is in the testing phase, where the software should be tested and verified in effort to find (and remove) any existing security vulnerabilities. For this we traditionally consider two main types of techniques: white box analysis and black box testing. Other techniques that combine characteristics of both are generally fall on the category of gray-box techniques.

2.3.1. White-box analysis

Consists in the analysis of the source code of the application without executing it. This can be done by manual code reviews and or inspections. Although it is known as the most effective technique to discover flaws in the code, it is also an extensive, exhausting and expensive way to address the problem. A less expensive alternative is to use automatic static code analyzers. Basically, a static code analyzer vets the source code searching for bug patterns that can lead to security flaws. For the specific example of SQL injection vulnerabilities, the analyzer would search for parts of code responsible to build SQL commands trying to identify if that part of code matches a pattern that allow to an attacker to modify the structure of an SQL statement.

There are open source white-box analysis tools available. One of them is FindBugs[11] that is a static code analyzer for java code, it looks into java binary code searching for bugs, which also includes bugs that can result in security breaches, as is our focus here. An advantage of this tool is that it is possible to integrate it with IDEs (for instance in NetBeans IDE [29]). IntelliJ IDEA[12] provides it's own on-the-fly static code analyses, this means that every time it recognizes a bug or a bad practice that can lead to a vulnerability the programmer is immediately warned to resolve the issue right away instead of perform an analyses over entire code at the end. The pitfall of this kind of tools is that its difficult to prove that a security flaw identified is an actual vulnerability, thus producing a high number of false positives.

2.3.2. Black-box testing

Also known as penetration testing consists in test for application vulnerabilities while executing them from an attacker point of view. This means that the tests are performed ignoring the internals of the service, with the requests being generated and submitted to the application. The decision of categorizing the service as vulnerable or not traditionally depends only on the analyses of the responses. Penetration testing can be seen as a specialization of robustness testing. Instead of assessing the robustness of system using unexpected inputs and analyzing how it impacts on system behavior, in the case of penetration testing the security of the system is assess by using malicious inputs. Regarding web services penetration testing is done by generating valid SOAP requests and send them to the server and keeping the responses. Afterwards, malicious messages (attacks) are generated using malicious parameters as input. These malicious requests are sent to the server too, analyze of the responses to malicious requests together with the responses to the non-malicious requests is used to classify services as vulnerable or non-vulnerable.

Several tools implementing penetration testing techniques are available, including some open source such as WSDigger [14] and WSFuzzer [15]. There are also available commercial tools that perform vulnerability scanning. The three tools that leads the market in web application security scanners are: HP WebInspect [16], IBM Rational AppScan [17], Acunetix Web Vulnerability Scanner [18].

Proposed by the researchers, the Improved Penetration Testing (IPT-WS)[22] is a tool that performs vulnerability scanners using a black-box approach, the reported vulnerabilities are the result of the responses to the requests.

Proposed by the researchers, the Improved Penetration Testing (IPT-WS)[22] approach starts with the generation of a representative set of non-malicious requests (workload), which is then submitted to the service and responses are stored. The next step is to generate a set of attacks (attackload) that consists in, for all operations, replace the valid inputs of the parameters by malicious inputs, one by one. The attacks used consist of a compilation of the attack sets used by several commercial and non-commercial tools, and attacks found in literature. Obviously this may generate a huge attackload that may cause the test to take too much time. In order to decrease its size, the workload requests are ranked and it is chosen a subset of these requests to generate the attacks. The ranking system obeys to well define rules in order to select the more representative subset as possible to the desirable size, according to the configuration. Once the attackload is created, the next step is to submit the requests of the attackload to the service. After that, well-defined rules are necessary to analyze the responses to attacks and to the workload. When necessary, robustness testing is also performed to improve the information available in order to find more vulnerabilities and reporting less false positives.

2.3.3. Gray-box testing

The limitation of black-box testing is that it is usually only based on the analysis of the outputs of services, while white-box analysis ignores how application behaves during a runtime attack. Gray-box testing is a hybrid between both approaches trying to overcome the limitations of both worlds while keeping their advantages. One advantage is that with gray-box solutions it is possible to detect vulnerabilities and attacks, since the code is being analyzed during the test. Even if the testing is performed in runtime it does not need t always access to the source code: sometimes, only by monitoring the interfaces of a service it is possible to improve the vulnerability detection process. The

Sign-WS [23] tool combines attack signatures with interface monitoring to detect more vulnerabilities while avoiding false positives. A special token is inserted in each attack to help in detection of vulnerabilities.

The approach proposed in [24] is a Runtime Anomaly Detection (RAD-WS) technique that relies on exercitation of services to create profiles, and verifying the existing queries on the profile during the attack phase.

AMNESIA is a gray-box approach that consists in statically analyzing of the source code looking for lines where SQL queries are built (hotspots) and then, for each hotspot, it is built a model of legit queries represented by NDFAs[19]. Later in runtime, the generated SQL command will be processed by a model checker that will confront it with the model of the respective hotspot. If the query doesn't reach a valid state of the automaton it is considered as an SQL Injection Attack and an exception is raised preventing the query to be sent to the database and reporting the attack to developers with information about the attack attempt. Otherwise, if the query structure ends in a valid state of the automaton the execution continues normally.

Acunetix WVS AcuSensor[20] is an Acunetix product that combines black-box testing with feedback gathered from sensors placed in the source code during execution. This application is only available to web applications developed in .NET and PHP. It doesn't require .NET source code since it can be injected in compiled .NET applications[20].

The Sign-WS [23] tool combines attack signatures with interface monitoring to detect more vulnerabilities while avoiding false positives. A special token is inserted in each attack to help in detection of vulnerabilities. The idea is to overcome the limitations of penetration testing by increasing the information available while, at the same time, keeping the process as close to black-box as possible. With these new attacks signatures it is possible to identify the input under test, the attackload is generated and submitted to the service under test. During the process of submitting the attackload a service monitor intercepts and captures all executed commands to the database and if a signature is found outside a literal string in a command it means that the parameter of the operation mapped by that signature is vulnerable.

The approach proposed in [24] is a Runtime Anomaly Detection (RAD-WS) technique that creates a profiles during the exercitation of service, the profiling consists in identify hotspots (parts of the code were SQL commands are executed) and store SQL commands intercepted during the web service exercitation. The result will be a set of profiles containing a list of valid SQL commands. During the attack phase, the SQL commands issued to the database are checked to see if they match the specific hotspot profile or not. This approach has two advantages: it can identify the vulnerable piece of the code and it can be used to prevent attacks since, we can identify an attack in runtime before its execution on database.

2.4. **Workload and Attackload Generation**

The workload or the test data represents the work that the system must perform during the experiments. In the specific case of services, the workload can be defined as a set of valid requests, i.e., is a set of non-malicious input parameters. In order to perform vulnerability detection using some of the techniques previously mentioned, web services need to be exercised in order to observe the outputs under normal conditions.

One method to generate workloads is to simple create SOAP messages with random values in the input parameters, of course keeping in mind the parameter data types, using the information available on the WSDL file. However it also possible to add information about parameter's domain in the descriptor file, despite the lack services that contains this information. This information will enhance the workload generation by allowing it to generate pseudo-random workloads. As we are able to know what values are allowed for each parameter (parameter: id data type: *int*, domain: 1 to 10) it is possible to generate workloads that are more adequate to the code under test.

Although the knowledge about parameter domains helps producing more effective workloads, it may be insufficient due to the fact that sometimes a domain of a parameter depends on the value of other parameters. In [30] an approach consists to add information about parameter domains using EDEL(Extended Domain Expression Language) in the XSD (XML Schema) associated to WSDL file, however programmers need to have this in mind when they are developing web services.

Other methods are proposed in literature, In [31] it is described an approach that generates workloads based on source code static analyses. This approach gives good warranties regarding code coverage though it needs access the source code, which is not possible in most of the cases. In [32] it is proposed a technique to automate test data generation based on state charts. Another possibility presented in [33] consists of the generation of workloads based on the characteristics of real-world workloads through the use of Markov Chains. While this option improves the representativeness of the generated workload, it needs from previous runs of the services.

One of the most important requirements when generating test data is code coverage. Code coverage is the degree on what our tests exercise the target base code [34]. This means that the generation of the test should be done in such a way that it will cover the most of the code. Imagine that a certain test data was generated with some input values that due to input validation, execution is always driven to exactly the same branch on the code. This represents a poor coverage of the tests generated that leads to lack of effectiveness of the tests. Code coverage is a main concern when testing web services, thus one of the key components identified on a vulnerability scanner is a workload generator. Another important property of the workload is its representativeness; this is, how close its characteristics are compared to a real workload.

Regarding attackloads it's important to say that are a set of requests were one parameter has been filled with a malicious input. The typical way of generating attackloads is base their generation on an existing workload injecting each attack in each parameter, one by one, thus the size of the attackload is: number of operations X number of parameter X number of attacks.

2.5. **Monitoring and Discovering**

Most of the vulnerability detection tools, especially the ones based on runtime analysis approaches rely on the analyses of data gathered by a monitor system, thus making monitoring a fundamental piece on any generic approach to detect vulnerabilities on web services. Although monitoring system is useful on vulnerability detection, it may be also useful in discovering new resources (e.g. other services, databases, servers, etc.), since the infrastructure under test may contain services that are not owned by the security performer, it may be possible that one or more services uses unknown services as resources, some monitoring approaches can help finding this 'hidden' services and enables their testing.

Some techniques for monitoring are reported in literature, in is described a service-based infrastructure monitoring system called Monere that aims to be is able to instrument relevant components across all layers of a service composition and to exploit the structure of BPEL (Business Process Execution Language) workflows to obtain structural cross-domain dependency graphs, also in [35] is proposed an approach of dynamic discovery of services in SOAs, this is collaborative approach that needs the service owner to share information about the services they manage in order to trace SOA evolution by automatically discovery of new services.

Another approach for monitoring is the use of code instrumentation; this approach consists on deployment of instrumented code in order to intercept calls to interesting parts of the code. With this approach a database driver can be instrumented in a way that is able to intercept calls to databases, thus the SQL command can be reported to a vulnerability detector and afterwards analyzed, at the same time we may possibly found unknown resources of the service-based infrastructure with this approach, in the case of a certain tester ignore the existence of such database, in the same way other software components can be instrumented in order to intercept calls to other web services enabling the discovery of unknown services when they are used by owned services and so on. Typically AOP (Aspect Oriented Programming) is used to code instrumentation, AOP is a programming paradigm decomposes a system in it basic functionalities and cross cutting aspects, then using a special language this functionalities can be captured and used to produce code [36].

2.6. **Existing/Concurrent Tools**

From the tools we investigated the most similar tools available at this moment are Hp WebInspect, Acunetix Web Vulnerability Scanner and IBM Security Appscan, these are the tools that we consider closer that what we pretend, they test services applying proprietary detection techniques and report the vulnerabilities to the users. Despite all of these tools do much more than web service vulnerability scans, our focus are only the features out web services because the intent of our tool is to scan and report web services vulnerabilities.

WSDigger is an open source tool designed by McAfee Foundstone that automatically performs black box testing in web services [14]. WSFuzzer it's a program written in Python that targets web service, and works very similarly to WSDigger, it automates some real-world penetration testing that are usually carried manually [15].

On other hand there are also available commercial tools that perform vulnerability scanning. The three tools that leads the market in web application security scanners are: HP WebInspect [16], IBM Rational AppScan [17], Acunetix Web Vulnerability Scanner [18].

HP WebInspect is *"an automated and configurable web application security and penetration testing tool that mimics real-world hacking techniques and attacks, enabling you to thoroughly analyze your complex web applications and services for security vulnerabilities. By enabling you to test web applications from development through production, efficiently manage test results and distribute security knowledge throughout your organization, WebInspect empowers you to protect your most vulnerable entry points from attack."* [16]

Acunetix Web Vulnerability Scanner (WVS) *"is an automated web application security testing tool that audits your web applications by checking for exploitable hacking vulnerabilities",* Acunetix WVS was first released in 2005 allowing vulnerability scans to web services since 2007, it is a easy to configure and easy to use to perform penetration tests on web services [18]. **Acunetix WVS AcuSensor**[20] is an Acunetix product that combines black-box testing with feedback gathered from sensors placed in the source code during execution. This application is only available to web applications developed in .NET and PHP. It doesn't require .NET source code since it can be injected in compiled .NET applications[20].

IBM Security AppScan [17] (before IBM Rational AppScan) is "*a leading application security testing suite designed to help manage vulnerability testing throughout the software development life cycle. IBM Security AppScan automates vulnerability assessments and scans and tests for all common web application vulnerabilities, including SQL-injection, cross-site scripting, buffer overflow, and flash/flex application and Web 2.0 exposure scans."*

Chapter 3
Tool Specification

This software has the objectives of combining all the necessary tools to perform effective web services vulnerability detection in a single tool, automatically gathering service information and discovering of new resources of the service-based infrastructures. The resulting tool must allow both consumers and providers to test the infrastructures that they are using with just some simple configuration. Also, it should allow that modules for specific operation such as workload and attackload generation could be plugged-in enabling an easy modification and improvement of the techniques used to perform vulnerability detection. Combination of such modules will also enable the possibility of creation of new vulnerability detection approaches and improvement of the existing.

This chapter analyses the specification of the tool that was developed to achieve these goals. This analysis starts with a *summary of the requirements* defined for the development of the tool. It is important to explain that the requirements were identified in high-level detail during meetings with the advisors. This way, the requirements are also influenced by years of working with automated tools for the detection of software vulnerabilities in web applications and services, and try to cope with the main difficulties faced when developing tools for that domain.

Then, it was necessary to analyze which requirements were already present in the existing vulnerability detection tools for web services. This *GAP Analysis* was performed during an extensive review of the related work and allowed us to understand which features were lacking on the state of the art and thus required the focus of our contribution. Finally, this analysis allowed us to specify the requirements in much thinner detail.

The next step was to define the architecture and the design to be followed that would allow us to a tool that fulfills these requirements and achieve the defined goals. First, it was necessary to select the architectural approach, and then establish a high level architecture. The last step was to define the detailed design and the interaction between the tool modules.

The structure of this chapter is as follows. Next section presents the summary of the requirement analysis. Section 3.2. presents the GAP Analysis. Finally, Section 3.3 presents the architecture and the design of the tool.

3.1. **Requirements Analysis**

To allow an easier read, we opted to provide only a summary of the requirements of the application. This summary is divided in the functional requirements relative to the general application and relative to the external modules and a set of non-functional requirements. The complete list of requirements for the application can be found on the annex “Software Requirements Specification for An Integrated Tool to Detect Vulnerabilities in Service-Based Infrastructures”[37].

Despite the usage of the tool is intended to be easy to configure and use, the fact is that some of the functionalities of the tools may not be available to every one, due to their access to systems to be tested or due to their lack of the require expertise to develop some of the possible usages. This way, we consider 3 main classes of users:

- **Service consumers** – can test the services that they want to use, but don't have internal access to the infrastructure to test the remaining services. Also, they have limited access on the internals of the services they can test;
- **Service providers** – can test the services that they provide and other services that they use of that their web services depend on. Take advantage of the capabilities of the tools for service discover and testing with different techniques according to the level of access they have on each service;
- **Advanced users** – can be either consumers or providers. What distinguishes them is their capability to develop new modules to extend the tool and create or improve vulnerability detection techniques. For this, they should be engineers, researches or people with skills in technology and specifically high expertise in the security field.

3.1.1. General Application

The tools shall allow the users to introduce a set of web services to scan by the *url*, it shall be possible to the user to select the access level he has to that service (*within-reach*, *partially under control* or *under control*), these services should be added to the infrastructure representation, the tool should get the list of the operations for each service introduced by the user. For each operation of each web service it shall be listed the parameter list. When its possible, the tools shall automatically get and show the information of *data types* and *domains* of parameter, such information shall be editable. The tool shall provide a way to configure the external modules to be used in the case of the three access types defined, the user shall be able select one of the available external module from a list for each a type of external module (workload generator, attackload generator, monitor and detecting), this shall be possible to do to the three access level profiles defined.

The users shall be able to select a service and start a test it or test all of the services of the infrastructure. The modules used during the scan of each service must be according with the service access level type, this means if a certain service is within reach, the tool should used the actual configuration (modules to be used) for within reach profile.

When the scan finishes a report shall be presented to the user, it should contain at least information about service with vulnerability, specifying the operation and parameter where the vulnerability was detected. It should provide as more information as possible depending on the technique used (e.g. line of code, attack used, typical request, query, etc.).

3.1.2. External Modules

In order to test and to validate the approach, it is necessary to demonstrate the composition of three different vulnerability detection techniques: IPT-WS, Sign-WS and RAD-WS. These techniques should all follow the same standardized and modular design and thus, each one should consist of the four external modules. It also should be possible to compose tools using different combinations of existing modules.

To achieve these goals, there is a set of modules that we are required to implement. These modules are:

- One random workload generator module, that shall get the information about the web services to generate the workload, i.e. a list of valid soap requests, and return it to the application.
- Two attackload generator modules, which after generating the attackload shall return it to the application.
    - One that shall generate attacks based on the workload and a set of attacks that must be injected in one parameter at a time;
    - One that shall generate attacks containing signatures that can univocally identify the operation and the parameter under attack;
- One monitoring system, based on probes that log information about the service and reports that information to the to the application. These probes should not modify the functional behavior of the system.
- Three external detection modules, which after classifying a parameter as vulnerable or not, report this information to the application.
    - One implementing IPT-WS, which shall classify based on the analysis of responses to workload, attackload and robustness testing;
    - One implementing Sign-WS, which shall classify based on finding or not active signatures in the queries received from the monitoring;
    - One implementing RAD-WS, which shall classify based on the anomalies found during the attack phase;

3.1.3. Non-functional requirements

There are some characteristics that the software to be implemented shall present. In order to add new techniques, it shall be possible to reuse common external modules implementing only the inexistent ones to complete the technique. Changes or additions on techniques shall not involve changes on the core module code or API. The techniques implemented in the tool shall produce similar results to the tools that implement each technique in ad-hoc fashion. The tools should be easy to configure and to use, it should not be needed to be a security expert user in order to perform the scans and interpret the reports.

These characteristics are reflected in a set of properties that the software must have:

- **Correctness** – The tool must operate correctly, it shall identify vulnerabilities present the same number of true vulnerabilities and false positives with a +/-5% deviation when compared with prototypes results when tested in the same conditions.

- **Interoperability** – It shall be possible to install and run the tool across all operating system on the market. Techniques using monitoring system shall only work with java-based web application servers, JBoss, Glassfish, etc.

- **Maintainability** – The tool shall be implemented and documented in a way that programmers that weren't involved during the implementation of the tool can easily perform bug fixes or modifications.

- **Reusability** – The implementation shall make use of software design patterns in order to easily enable the evolving of the tool and to facilitate integration of this tool with other application.

3.2. GAP Analysis

Although there are some open source and commercial tools for vulnerability detection in web services, these tools were either developed in an ad-hoc fashion or have a closed architecture that does not allow third parties to extend it thus missing some of the features present in our tools.

The commercial tools like HP WebInspect[16], IBM Rational AppScan [17], Acunetix Web Vulnerability Scanner [18] do not support third party extension since they have closed architectures, they do not follow standard architectures and the vulnerability detection technique used is the same independently of the access the test carrier has to the services.

On other hand some of tools open source tools WSDigger [14] and WSFuzzer [15] supports the addition of plugins. However, such plugins do not add nothing relevant to the way the services are tested, they just allow users to add different attacks to perform penetration tests. The main problem of these tools remains: they are not vulnerability scanners, i.e. they do not have the ability to detect vulnerabilities. Instead, these tools test the web services with the malicious inputs and log the results, and afterwards the user should have the expertise and the huge task of analyzing the information searching for vulnerabilities.

To summarize, the innovation required to achieve the goals of these work, when compared with the current state-of-the-art tools, i.e., the features required in our tool that cannot be found in the existing tools, are:

- The possibility of advanced users to extend the tool with new and improved modules, thus improving existing vulnerability detection techniques;
- The possibility of advanced users to extend the tool with their own modules, thus enabling and facilitating the development of new vulnerability detection techniques;
- The possibility to classify services by the access level that users have to them and the creation of profiles for each of the service type to dynamically change the technique to be used for each case.
- The possibility of manually provide information about resources and services and their relationships;
- The ability to discovering information about resources and services automatically, to extend the scanning process;

3.3. Tool Architecture

In this work we created an integrated tool in which functionality can be inserted and/or modified with a low effort, in a way that allows a single tool to implement multiple vulnerability detection approaches. A key problem on assessing service-based infrastructures security is that different scenarios require different vulnerability detection approaches. In fact, the use of several tools is typically needed in order to increase the tests coverage and thus improve the number of vulnerabilities detected.

3.3.1. Modular Approach

After the requirements analysis it becomes obvious that the main non-functional requirements were extensibility and maintainability together with the fact that the external components of the system should be decoupled for the remaining external components of the system. All of these requirements have driven us to the paradigm of modular programing or **modular design**.

A human strategy to deal with complex problem is to divide it in small problems that all together solve the whole problem. However that is not an issue to a machine, even if some program is coded in small parts a compiler will transform it and execute it in a sequential way. So, we can say that *modularization is helpful from a human perspective* because it facilitate the comprehension of the entire problem thus reducing cost on maintainability, extensibility of a product.

We can find several definitions for modularity among the literature and although in [] modularity is defined in a very generic way, we have found that it goes in our requirements direction. In [38] module is defined as *"... a unit whose structural elements are powerfully connected between themselves and relatively weakly connected to other units..."*. This definition brings up idea of interdependence within modules or dependency across modules. In other words modules are units that are structurally independent from each other, but work together.

One concept that many times appears side-by-side with modularity is the concept of *separation of concerns*. The definition of concern is hard and usually very vague as "any consideration... about the implementation of a program" or "any matter of interest in a software system". However, Ossher and Tarr define a concern as "*…a part of a software system that is relevant to a specific concept or purpose…"* which fits better on our context because we want to separate our modules by purpose. Separation of concerns allows the locality of different kinds of information in the programs, making them easier to write, understand, reuse, and modify [39]. These attributes coincide with our tool main goal, and one way of achieve this properties is modularization.

In conclusion, on our scope a good definition for module would be: an independent software component that contains all elements and processes needed to the realization of a given concern, and provide functionality related with that concern through an interface.

3.3.2. Architecture Overview

The design of the integrated tool based on a modular design started from a very basic approach. Naturally, in the big picture three main architectural elements were easily identified as shown in **Figure 2** an application core, an API, and the several external modules.

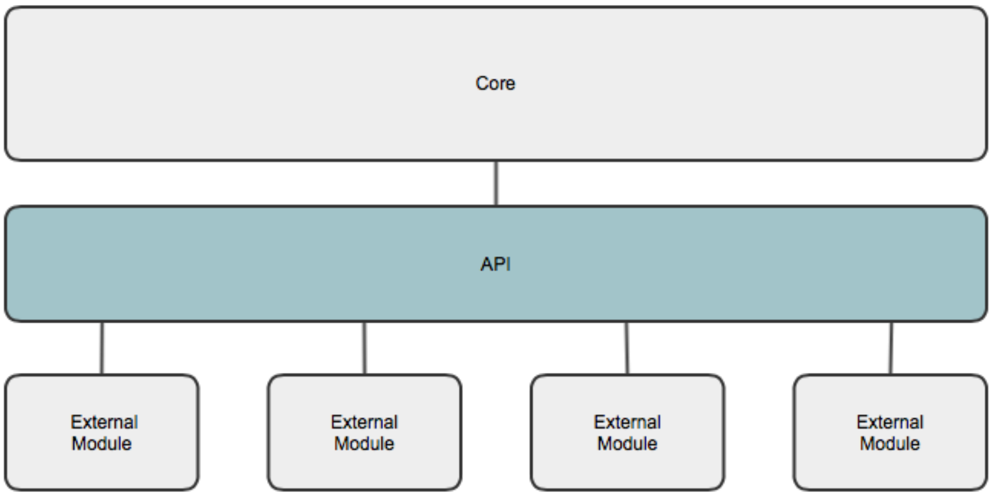


Figure 2 – High-level Architecture

The reasoning behind this overall architecture is very simple: in these types of tools there are several functionalities that are common and at the same time some other functionalities are technique specific. It was clear that it was necessary to decouple the specific functions that needed to be present in almost every vulnerability detection techniques. Thus, we decided to provide these functionalities as external modules. In the case of the generic functionalities shared across all of the techniques, we decide to combine and contain them in the same architectural element. The final element, essential to allow the architecture to work, is the application-programming interface (API). The API plays a very important role in this architectural model, since it contains all of the elements that must to be known from core and external modules as well as the interfaces provided from both to empower interaction.

3.3.3. Detailed Design

After the overview of the architecture, it is important to detail on each one of the architectural elements presented and describe its main components. **Figure 3** depicts the overall design of the application.

As it is possible to observe, the core module consists of eight main components. On the one hand, four of this components only uses and provide interfaces to components that belongs to the core module, that components are: ModuleLoader, ServiceLoader, Interactor, Discovery. On the other hand, the other four components interact with internal components of the core module and at the same time they communicate to the external modules. The following paragraphs present each one of these modules.

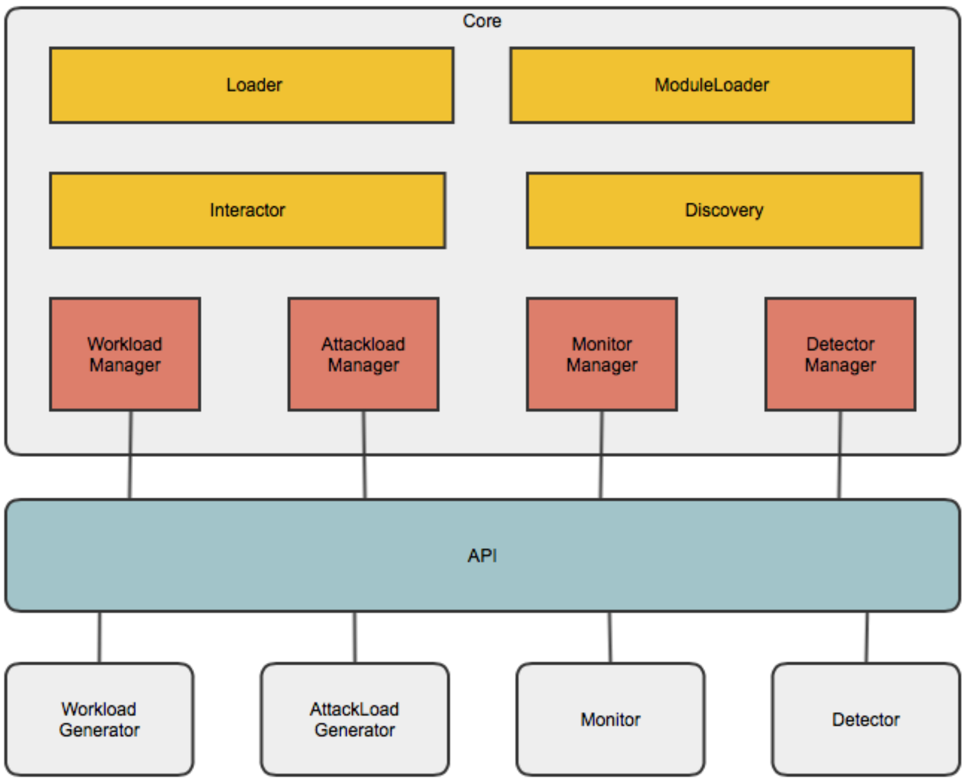


Figure 3 – Detailed Architecture.

ModuleLoader scans for new external modules and loads them when the application starts, it will look for the modules in a specific folder, it will load every class that extends Module class ignoring all others. It detects the kind of modules a make them available for use by the respective manager components.

The **Loader** component is responsible to get all the possible information the web services, namely service operation, operation parameter, parameter data type and domain with the given.

In tools based on testing like this, it is always mandatory to interact with the services under test. That is the **Interactor** component responsibility. This component contains a pool of threads that, using a third party library, can submit SOAP requests and wait for the respective response, this information is kept in the adequate objects to be used during the detection phase if needed.

When monitoring is involved, it is possible to obtain messages that contains information about new resources previously unknown that also makes part of the service-oriented infrastructure. These messages need to be handled asynchronously. To execute that task, a component named **Discovery** was implemented, it receives the messages related to discovery and new Resources are created and added to the resource list of the application.

WorkloadManager is the module that is responsible to choose the correct external module to generate the workload for the service under test depending on the access level the users has to it. Also, all of the operations related to workload are provided by this component. Even the workload requests to be submitted to the services, are previously requested to this module, which will ask to the external workload module to create the workload requests. Afterwards, this module submits them through the Interactor.

The role of the **AttackloadManager** is analogue to the **WorkloadManager**, with the difference that it is the responsible to get the proper external attackload generator. The rest of the operation provided by this component is similar to the WorkloadManager but this time for malicious requests.

The **MonitorManager** is the central of the messages obtained from the external monitor modules. This component filters the messages by type and forwards them to the correct destination. Also, all the operations related with monitoring, like stop/start the external monitoring are located in this component.

The last and the most complex component is the **DetectingManager** component. It is similar to the last three, it will interact with the external module configured to the service under test type, but it is also provides an interface with several operation that allows the external modules to request action. This component in particular interacts with all of the components that manages external module, replicating their action and providing them to the external detecting module. The reasoning behind this is that the external detection module drives the scanning process, thus it needs to ask to the DetectingManager to forward action requests from the external detecting module to the internal components.

3.3.4. Module Interaction

Figure 4 shows how the system components interact each other. First of all the application needs a component responsible for obtaining information about the services under testing. This, provided by the user, includes information about operations, list of parameters of each operation and the respective data types and domains. Such information will be used by the discovery component, which will be responsible for add new resources found in runtime. Both with this component and the information provided by user manual input he goal is to create and maintain an architectural model of the infrastructure, which is a representation of the services, resources and their relationships.

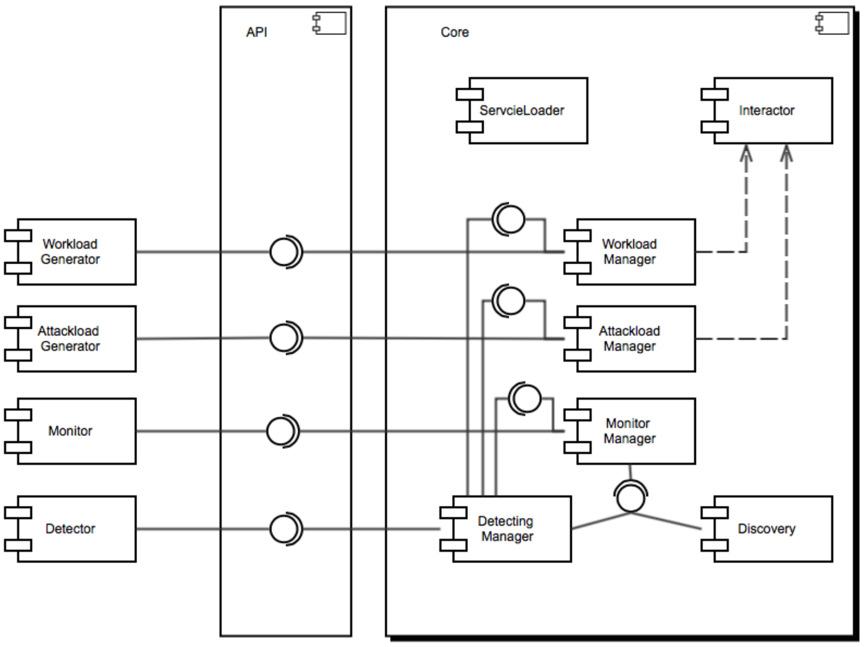


Figure 4 – System Component Diagram

The component responsible for managing the workload generator will make such information available to the external *Workload Generator*, and it will receive the generated workloads from the external module this component also provides operation to the DetectingManager through an Interface. Some techniques will also require workload manager to submit the workload to the services and store the responses, despite the call to the function to submit request is made to the WorkloadManager, in fact only the soap requests are built in this module an then provided to Interactor component, this component is actually the one that effectively sends such request to web application servers.

The Attack Manager component provides the workload if it has been generated before to an external *Attackload Generator* and receive the attackload generated by the external module. Similarly to the workload manager it provides interfaces to the Discovery module that can requires this component to generate and submit attackloads which in turn asks to the Interactor to perform the submission of the soap requests containing attacks.

A monitoring manager component manages the interaction with an external *Monitoring System*. This component provides an interface to the external monitoring module that pushes messages to the MonitoringManager existing queue, which in turn filters and pushes messages to the discovery component and to the Detecting Manager component because some of the vulnerability detection techniques make use of additional information coming from monitoring. This module also provides an interface in order to DetectingManager have some control about monitoring.

The last component is the DetectionManager, it interacts with the external *Detecting Module*, and the external module receives all the information existing about the service under test and provides an interface that allows the DetectingManager to demand the external module to initiate the detection process. In the external detecting module is defined the workflow of the desired detection technique, for that purpose the

DetectingManager also provides an interface that contains the all the operations that allow the control of all of the required internal components.

All of the shared information and the interfaces need to the core components and external components are contained inside an API component. This way all the information required to the implementation of this modules are defined in this component, thus making the API component the unique dependency of the external modules.

Chapter 4
Implementation

This chapter describes the implementation of the software specified in Chapter 3 As specified, the tool is divided in three main modules: the application core, the API, and several external modules (listed in the requirement of subsection 3.1.2. . For each of the presented modules, it is described the main classes and the most relevant methods. Simplified class diagrams are also presented and discussed to better understand the implementation decisions during development.

It was decided to present in first place the implementation detail of the API module, as the concepts that it introduces are of utmost importance for the reader to understand both the core module and the external modules implemented. In practice, the API specifies all the objects and methods that are shared between the core module and each one of the external modules.

After that we present the core module that implements the functionalities that are invariant across the multiple vulnerability detection techniques. This module also makes an effort to concentrate the crosscutting concerns, to increase the modularity of the remaining parts of the implementation.

Finally, we present the implementation of modules necessary to configure the three vulnerability detection approaches (as listed in Subsection 3.1.2.). The section presents the main implementation decisions for these modules and what is necessary to build the three techniques from them.

The structure of this chapter is as follows. Next section presents the implementation of the API. Section 4.2. presents the implementation details of the Core module. Finally Section 4.3. presents the implementation of the external modules.

4.1. **API**

The *Application Programming Interface* built and made available for future developers can be seen as a library that contain all the Abstract classes for the implementation of the external modules. These abstract classes also contain abstract methods that any developer should implement together with the required interfaces between all the external modules and the core module. As the complete class diagrams is too big and too complex, we decided to analyze figures with the most important parts of a simplified version of the class diagram. **Figure 5** shows the class diagram of the abstract classes that external modules shall implement.

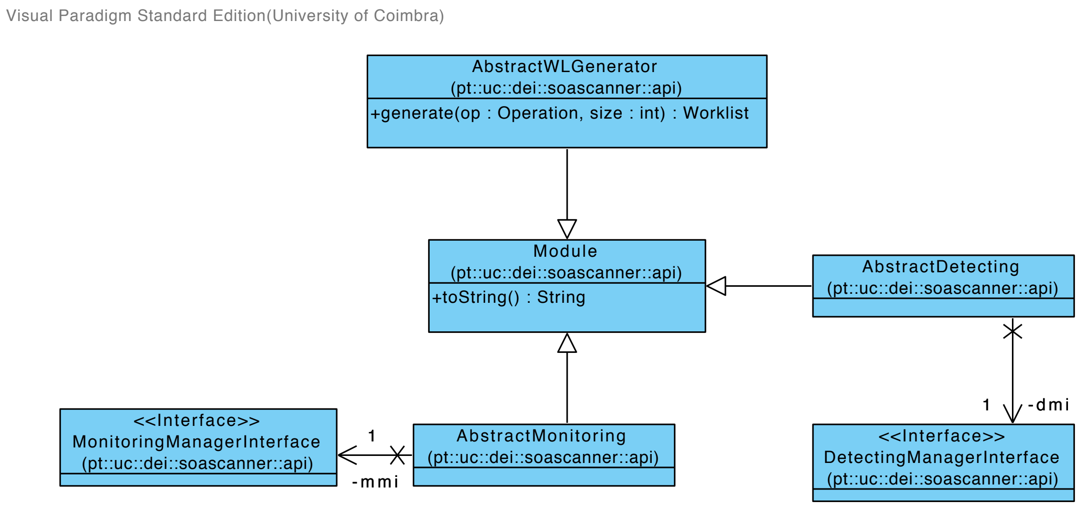

Visual Paradigm Standard Edition(University of Coimbra)
AbstractWLGenerator
(pt::uc::dei::soascanner::api)
+generate(op : Operation, size : int) : Worklist
Module
(pt::uc::dei::soascanner::api)
+toString() : String
AbstractDetecting
(pt::uc::dei::soascanner::api)
1
-dmi
<<Interface>>
MonitoringManagerInterface
(pt::uc::dei::soascanner::api)
1
-mmi
AbstractMonitoring
(pt::uc::dei::soascanner::api)
<<Interface>>
DetectingManagerInterface
(pt::uc::dei::soascanner::api)

Figure 5 – Class diagram for external modules Abstract classes.

Additionally, all of the classes from both core and external modules need to be aware about the classes used to represents the elements of the infrastructure. Thus, those classes should be shared and also present on the API. These classes are depicted in **Figure 6** to better understanding. The classes that used to represent the infrastructure elements are the following:

- Resource
- Service
- Operation
- Field
- Relation

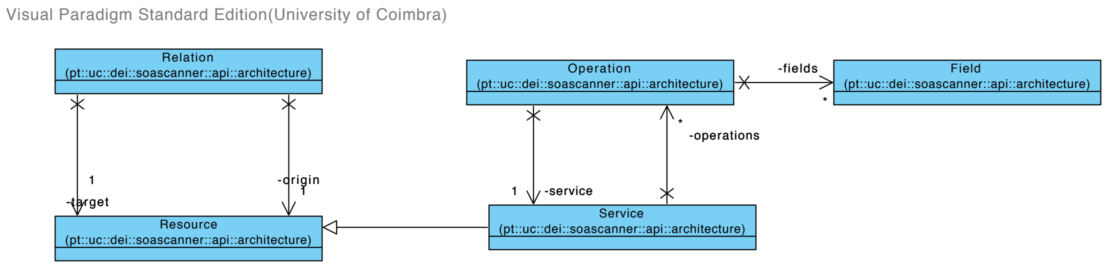

Visual Paradigm Standard Edition(University of Coimbra)
Relation
(pt::uc::dei::soascanner::api::architecture)
Operation
(pt::uc::dei::soascanner::api::architecture)
-fields
*
Field
(pt::uc::dei::soascanner::api::architecture)
*
-operations
1
-target
-origin
1
1
-service
Resource
(pt::uc::dei::soascanner::api::architecture)
Service
(pt::uc::dei::soascanner::api::architecture)

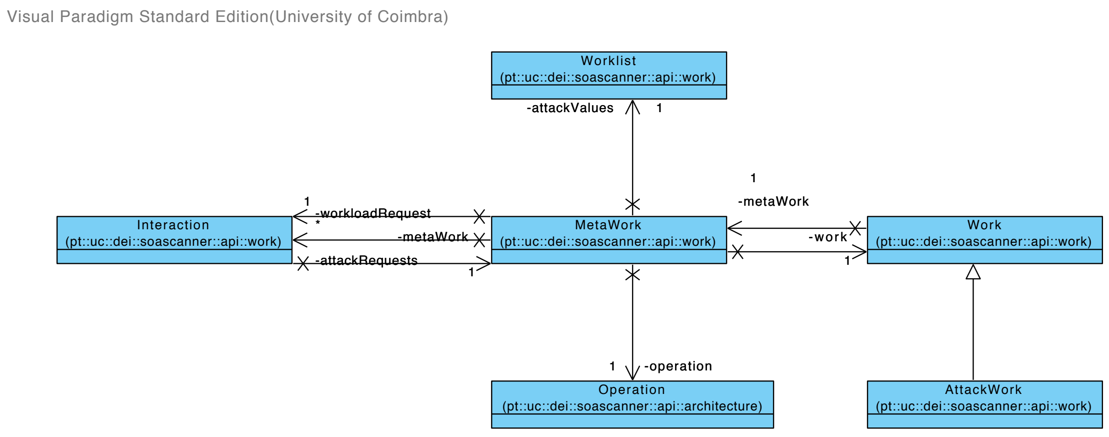
Figure 6 – Class diagram for objects used to represent the infrastructure elements.

As it is possible to observe, a *Service* is a specialization of a *Resource* and a *Service* is composed by a list of *Operation*, while an *Operation* contains a list of *Field*. A *Relation* defines an ordained pair of *Resources* called *origin* and *target*.

Meanwhile, an instance of *Operation* has also its attributes instances of classes that address workloads and attackload generated for that specific *Operation* (such relations are depicted in **Figure 7**). List of the main classes:

- Work
- AttackWork
- Worklist
- Interaction

- MetaWork
- MetaWorkMap



Visual Paradigm Standard Edition(University of Coimbra)
Worklist (pt::uc::dei::soascanner::api::work)
-attackValues
1
Interaction (pt::uc::dei::soascanner::api::work)
-workloadRequest
-metaWork
-attackRequests
MetaWork (pt::uc::dei::soascanner::api::work)
-metaWork
-work
Work (pt::uc::dei::soascanner::api::work)
-operation
Operation (pt::uc::dei::soascanner::api::architecture)
AttackWork (pt::uc::dei::soascanner::api::work)

Figure 7 – Class diagram for objects related to Workload and Attackload generation.

In practice, a *Work* is no more than an ordained list of string that will be later used as values for the parameters of the SOAP request. Thus, the length of the list is equal to the number of *Field* of the specific *Operation* under test. Such *Operation* stores all of the *Work* that composes a workload. An *AttackWork* is similar to a *Work* but it has one of the parameters mutated with an attack thus, an *AttackWork* is a specialization of *Work* and inherits the attributes of *Work* but in addition it as a new attribute that is the field *id*, which identifies the parameter under attack. A *Worklist* consists on a list of *Work* (or *AttackWork*) and it has as attribute also a *size*.

The goal of the class *Interaction* is to keep all the information related to the result of a request corresponding to a determined work when submitted to the service. When an *Interaction* is instantiated it requires as parameters a *MetaWork* and string with the request content. It has also a second constructor that requires the field id in addition, for the case that such interaction is related with an attack and is used to identify the parameter under attack in the respective Interaction.

A *MetaWork* is a class that was implemented to keep the information about an Interaction relative to a specific *Work* and a list of the Interaction resulting of the attacks generated based on that *Work*. It contains a reference to the *Operation* that the specific *Work* is part, which is done to facilitate the navigation between objects. A List with all the *AttackWork* generated based on the *Work* in this *MetaWork* is also kept, as well as the vulnerabilities found during vulnerability detection, which is added during the scan process.

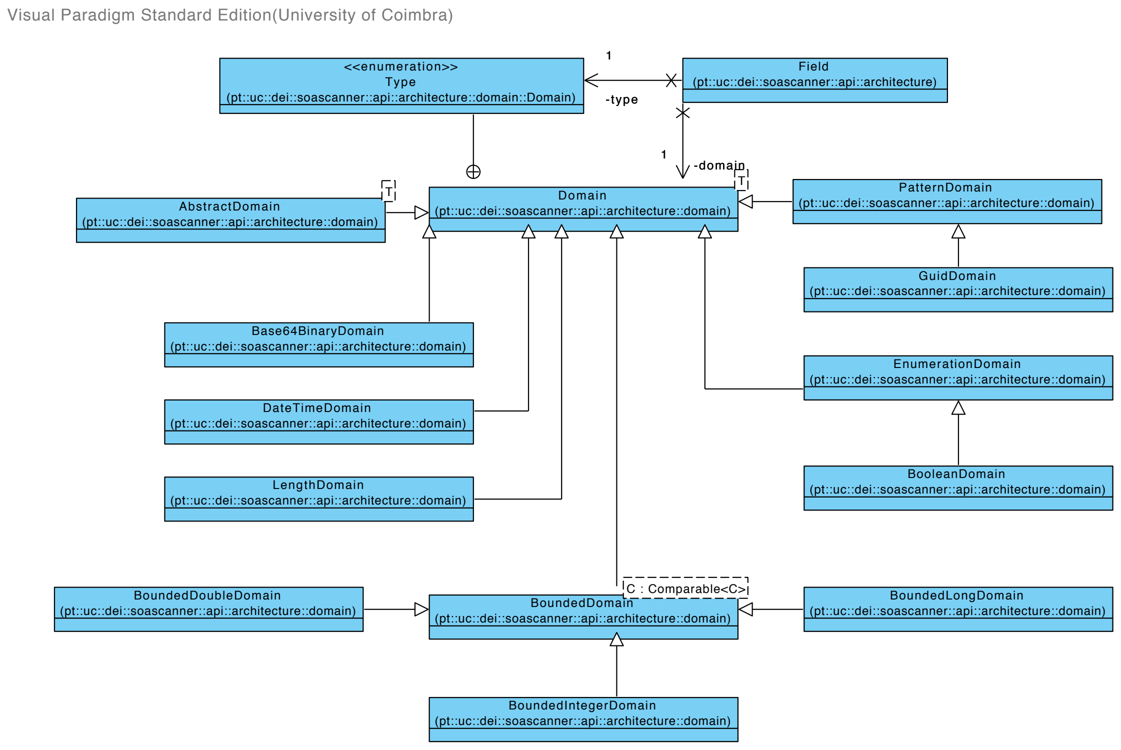
To facilitate the access to these *MetaWork* objects, a *MetaWorkMap* was implemented. A *MetaWorkMap* is an extended *HashMap* where the key is the *Operation* and the value is a list of *MetaWork* objects. Each *Service* instance, which represents a service under test, has as an attribute a *MetaWorkMap*.

As mentioned throughout this document, one aspect of utmost importance is to keep information about fields (parameter) of the operations, namely information such data types and domains. To allow this, each of the instances of Field contains a domain, which is represented by the classes: *Domain* and *AbstractDomain.*

The information about data types is done together with the definition of the domain. Also, the domains for most of the data types can be bounded, length or enumeration. However, for some data types certain domains do not make sense, as is the case of length domains that cannot be applied to a *Boolean* type because a *Boolean* type only admits either true or false. Thus, each *Field* instance has one instance of the following classes as an attribute:

- DateTimeDomain
- BooleanDomain
- BoundedDoubleDomain
- BoundedIntegerDomain
- BoundedLongDomain
- EnumerationDomain
- LengthDomain

The existing classes used to represent the Field data types, domains and the respective relation are presented in Figure 8.



Visual Paradigm Standard Edition(University of Coimbra)
<<enumeration>>
Type
(pt::uc::dei::soascanner::api::architecture::domain::Domain)
1
-type
Field
(pt::uc::dei::soascanner::api::architecture)
1
-domain
T
AbstractDomain
(pt::uc::dei::soascanner::api::architecture::domain)
T
Domain
(pt::uc::dei::soascanner::api::architecture::domain)
PatternDomain
(pt::uc::dei::soascanner::api::architecture::domain)
GuidDomain
(pt::uc::dei::soascanner::api::architecture::domain)
Base64BinaryDomain
(pt::uc::dei::soascanner::api::architecture::domain)
EnumerationDomain
(pt::uc::dei::soascanner::api::architecture::domain)
DateTimeDomain
(pt::uc::dei::soascanner::api::architecture::domain)
LengthDomain
(pt::uc::dei::soascanner::api::architecture::domain)
BooleanDomain
(pt::uc::dei::soascanner::api::architecture::domain)
C : Comparable<C>
BoundedDoubleDomain
(pt::uc::dei::soascanner::api::architecture::domain)
BoundedDomain
(pt::uc::dei::soascanner::api::architecture::domain)
BoundedLongDomain
(pt::uc::dei::soascanner::api::architecture::domain)
BoundedIntegerDomain
(pt::uc::dei::soascanner::api::architecture::domain)

Figure 8 – Class diagram for objects relatives to field data types and domains.

The messages reported from the external modules to the core module are also defined in the API: *Message, DiscoveryMessage and MonitorMessage*. The **Figure 9** portrays the relations between these classes.

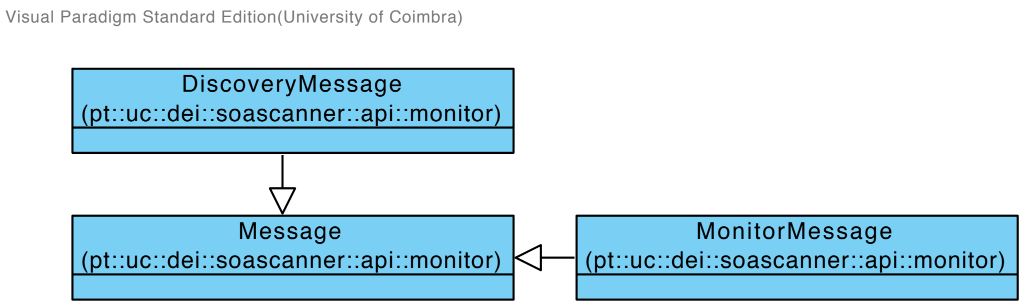

Visual Paradigm Standard Edition(University of Coimbra)
DiscoveryMessage
(pt::uc::dei::soascanner::api::monitor)
Message
(pt::uc::dei::soascanner::api::monitor)
MonitorMessage
(pt::uc::dei::soascanner::api::monitor)

Figure 9 – Class Diagram for objects related to Messages.

The *Message* class has *Type* as an attribute. The *DiscoveryMessage* and the *MonitorMessage* are specializations of the class Message. The DiscoveryMessage contains the information about resources discovered during monitoring while the MonitorMessage reports information about the commands that was executed in the in the resources related to vulnerabilities (e.g. database queries, etc.) during the request submitting and in some cases additional information.

4.2. **Core Module**

This section presents the most relevant details of the core module implementation. The main responsibilities of the core module are to address all common concerns among the various vulnerability detection techniques (e.g. load services, get web services information, create and submit request) plus it contains all the managers responsible for interacting with external modules. The **Figure 10** presents a simplified version of the class diagram of the core modules, which is described in the following paragraphs.

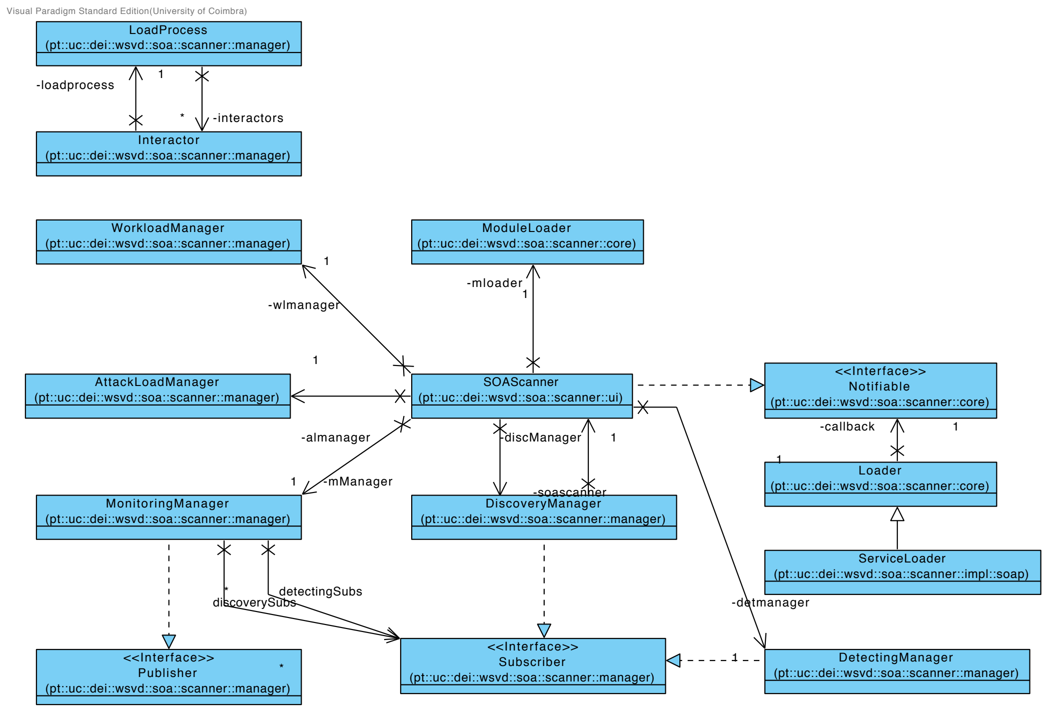

Visual Paradigm Standard Edition(University of Coimbra)
LoadProcess (pt::uc::dei::wsvd::soa::scanner::manager)
-loadprocess
1
*
-interactors
Interactor (pt::uc::dei::wsvd::soa::scanner::manager)
WorkloadManager (pt::uc::dei::wsvd::soa::scanner::manager)
ModuleLoader (pt::uc::dei::wsvd::soa::scanner::core)
-mloader
-wlmanager
AttackLoadManager (pt::uc::dei::wsvd::soa::scanner::manager)
SOAScanner (pt::uc::dei::wsvd::soa::scanner::ui)
<<Interface>> Notifiable (pt::uc::dei::wsvd::soa::scanner::core)
-callback
Loader (pt::uc::dei::wsvd::soa::scanner::core)
ServiceLoader (pt::uc::dei::wsvd::soa::scanner::impl::soap)
-almanager
-mManager
-discManager
-soascanner
MonitoringManager (pt::uc::dei::wsvd::soa::scanner::manager)
DiscoveryManager (pt::uc::dei::wsvd::soa::scanner::manager)
detectingSubs
discoverySubs
-detmanager
<<Interface>> Publisher (pt::uc::dei::wsvd::soa::scanner::manager)
<<Interface>> Subscriber (pt::uc::dei::wsvd::soa::scanner::manager)
DetectingManager (pt::uc::dei::wsvd::soa::scanner::manager)

Figure 10 – Class diagram for the core module of the application.

SoaScanner is the main class and contains all the instances of the required classes and data structures to keep important data during the scans and all the control of the application. This class contains all GUI event handlers that will instantiate and initialize all the objects and structures needed to perform the required actions.

ServiceLoader class extends the abstract class *Loader*, this abstract class is a thread that contains an abstract method *load()*. This class is a thread that extracts the information about services from a *url* string, also it is responsible to discover operations, parameter. Methods to automatically set the parameter data type and domain are implemented inside this class. The main methods of this class are:

```
//function uses the soapUI library to extract all the information about the services.
@Override
public Result load(){}

//method that obtains the fields information, data types and domains
//this method is called by load(), this method fills all the parameter list of the
//service operations, including data types and domains.
private String processRequest(){}
```

LoadProcess is the class that manages the requests to be made to the services. An instance of this class is created for each operation we want to submit requests, it also uses a pool of threads the type *Interactor* the main methods this class are:

```
//receives and Interaction instance and add it to a queue
enqueueInteraction(Interaction interaction){}

//waits until of the threads Interactor finishes their work
awaitTermination(){}
```

Interactor is a *Thread* that takes interactions from the queue of *LoadProcess* class *Interaction* queue (implemented using a *BlockingQueue*) and submits the requests to server. Finally it sets the response on the Interaction object. The main method of the class is:

```
//receives an Interaction submit the respective request, waits for the response to the
//request and set it on the response content to the Interaction object, if the
//interaction was succeeded it returns true, otherwise returns false;
private Boolean interact(Interaction interaction){}
```

WorkloadManager is the class that manages all the operations related with workload generation, which are implement in the methods presented next. It contains the information about the selected external generators for each type of service, thus when a workload is required the external module to use is already known by the instance of this class.

```
//Receives an instance of class Service and get the workload for each service operation,
//here is called the method generate() from the external workload generator.
public boolean getWorkload(Service service){}

//exactly the same goal as the last method, but this time for a specific operation
public boolean getWorkload(Operation op){}

//Receives a SoapService instance as parameter, for each operation its created a new
//instance of LoadProcess class is instantiated and for each work a new Interaction is
//created and added to the LoadProcess Interaction queue, this methods finishes when the
//Loadprocess method awaitTermination() finishes.
public boolean submitSoapWorkload(SoapService soapservice){}


//exactly the same as the method described before, but for a specific operation.
boolean submitSoapWorkload(SoapService soapService, Operation operation){}


//Receives the Operation object and a work, returns a string that corresponds to the
//soap request content.
private String buildBaseRequest(Operation op, Work work){}
```

Similarly to WorkloadManager, the *AttackloadManager* class manages all the functionality about attackload generation. It is the class that manages all the operations related with attack generation. Also, it contains the information about the selected external generators for each type of service, thus when an attackload is required the external module to use is already known by the instance of this class:

```
//Receives an instance of class Service and get the attack for each service operation,
//here is called the method generate() from the external attack generator.
public boolean getAttackLoad(Service service){}

//Similar to the previous one, but this time only generates the workload for a specific
//operation o the service
void getAttackLoad(Operation operation){}

//Receives a SoapService instance as parameter, for each operation its created a new
//instance of LoadProcess class is instantiated and for each work a new Interaction is
//created and added to the LoadProcess Interaction queue, this methods finishes when the
//Loadprocess method awaitTermination() finishes.
public boolean submitSoapAttackload(SoapService soapservice){}

// Receives a SoapService, an operation and a field, and for that specific operation a
//LoadProces is instantiated, and the interaction based on attacks injected on the
//specific field are created and enqueued on the LoadProcess. This method finishes when
//all the requests have been submitted.
public boolean submitSoapAttackload(SoapService soapservice, Operation operation, int
field){}

//Receives a SoapService, an Operation a work and a Field. For that a
//operation a LoadProces is instantiated, the interactions based on the provided work
//are created and associated to the provided field and enqueued on the LoadProcess. This
//method finishes when all the requests have been submitted.
protected List<Interaction> submitOnDemand(SoapService soapservice, Operation operation,
Worklist wlist, int fieldid){}
```

The *MonitorManager* class also extends a thread and it is in charge of receive, filter and forward messages from the external monitor to other internal component; it also contains the methods to control the external monitoring. This class implements the Publisher interface and the *MonitoringManagerInterface*. The most relevant methods of this class are:

```
//this methods calls the method start from the external module to be used for the
//particular ServiceType, and provides its interface to the external monitor.
public boolean startMonitoring(ServiceType serviceType){}

//Is the implementation of the method defined in the MonitorManagerInterface, the
//external monitors have this interfaces and they class this method to report messages
to //the MonitorManager, this methods add the received message from the external module
to //a queue;
public void pushMessage(Message msg){}

//this methods are defined in the Publisher interface, that is available only to
//internal components, its called when an internal component wants to start receive
//messages from the MonitoringManager. When a Subscriber wants to register it will be
//added to a subscriber set depending on e type of message it pretends to receive.
public void register(Subscriber s, Message.Type type){}

//removes the subscriber from the subscriber set for the specific message type.
public void unregister(Subsdcriber s, Message.Type type){}

//get messages from the complete message queue, checks the message type, and forward the
//message to the components that subscribed that message type
void run(){}
```

DiscoveryManager is a thread that implements the *Subscriber* interface. This class subscribes the message types related with the discovering of new resources, the *MonitorMessage* insert inserts messages to this class, this class will manage that messages and add the new resources to the resource set present on *SoaScanner* class. The main methods are:

```
//method defined on the Subscriber interface, this method is called by a Publisher and
//passing as parameter a message. In this class the specific implementation of this
//method consists in adding this message to a queue.
@Override
    public void update(Message msg){}

//process the message that contains information from new database found and add it to
//the resources set
private void addDatabase(Message msg){}

//process the message that contains information from new service found and add it to the
//resources set
private void addService(Message msg){}


//get messages from the queue, and calls the methods that handles the message depending
//its type
@Override
void run();
```

The class that provides all the methods needed to the external detector to perform its tasks is the *DetectingManager*, this class implements the *DetectingManagerInterface* and also the *Subscriber* interface. The reasoning behind this, is that it that sometimes it needs to subscribe to the *MonitoringManager* class in order to receive the messages related with the detecting information, and at the same time it provides an interface with the operations needed to the external detecting module drive the test. The main methods in this class are the following:

```
//calls the external module to start the test for ta specific service
    public boolean  analyze(Service service) {}

//asks for the WorkLoadManager to generate the workload for the given service
    @Override
    public void generateWorkload(Service service) {}


 //asks for the WorkLoadManager to generate the workload for the given operation
    @Override
    public void generateWorkload(Operation op) {}

// asks to the WorkloadManager to submit the workload for all operations of a given
//service
    @Override
    public void submitWorkload(Service service) {}

//asks the Workload generator to generate the workload to a specific operation of the
//given service
        @Override
    public void submitWorkload(Service service, Operation operation) {}

//requests to AttakloadManager to generate the attack to the given service
    @Override
    public void generateAttackLoad(Service service) {}

//Request to AttackLoadManager to generate the attackload for a specific operation oa a
//given service
        @Override
    public void generateAttackLoad(Operation operation) {}

//request the AttackloadManager to submit the attackload of a given service
    @Override
    public boolean submitAttackLoad(Service service) {}

//requests the MonitorintManager to start the monitoring chosen for a specific service
//type
    @Override
    public boolean startMonitoring(Service.ServiceType serviceType) {}

//Requests the AttackLoadManager to submit the attackload for a specific parameter of an
//operation that belongs to a given service
    @Override
    public boolean submitAttackLoad(Service service, Operation operation, int field) {}

//creates a new Instance o ReportView class with the results of the scan an shows it in
the user in a GUI
    @Override
    public void displayReport(Service service) {}

//method called by the Publisher with a new Message, this method will push the message
//to a queue in the external discover module through a provided interface
    @Override
    public void update(Message msg) {}

//registers to the Publisher, this is register itself to the MonitoringManager in order
//to receive messages from it.
    @Override
    public void register(AbstractDetecting aThis) {}

//remove itself from the subscriber list of the MonitorManager
    @Override
    public void unregister(AbstractDetecting aThis) {}

//request the WorkloadManager to submit the workload of a giver operation
    @Override
    public void submitWorkload(Operation operation) {}
```

4.3. **External Modules Implemented**

In order to test and validate our tool's architecture, several external modules were implemented. As explained previously, the implemented modules were the ones necessary to enable the composition of three techniques that can perform the three service control levels respectively. The Improved Penetration Testing (IPT-WS) to scan *within reach* services, the Signature-based Scan (Sign-WS) to scan *partially under control* services and Runtime Anomaly Detection (RAD) to scan *under control* services. In the following sections will describe the implemented modules and how can we build this three techniques from that available modules

4.3.1. Workload Generator Module

The only workload generator module implemented was a **Random Workload Generator.**

Although it might not seem so, this module is the one that most affects the effectiveness of the techniques because all of the requests to send to the services are based on the workload generated by this module. Lack of information about fields' domains results in non-representativeness, which may affect the coverage of the workload generated as we can see in the following example:

```
If(parameter1>10){
    executeQuery(query);
}else{
    return;
}
```

Imagine that this is a piece of code inside a web service operation implementation, and that *parameter1* is given as an input of the operation. In this situation if our workload does not contain a value to *parameter1* greater than 10 then the query will never execute and a possible vulnerability will remain unreported.

Also, one of the advantages of this integrated tool can be partially explained with the fact that all of the three techniques use this workload generator module. This modules provides a function that receives the information about the fields of an operation and a workload size, after, this module uses the information about the fields (data type and domain) and create a random set of non-repeated works until it reaches the preferred workload size.

4.3.2. Attackload Generators Module

The attackload generator provided in this tool is based on the previously generated workload. Although this is not mandatory our specific techniques we require the previous generation of a workload. Two out of three techniques use the same attackload generator, this means we can reuse the modules in different techniques.

The **Regular Attackload Generator** module assumes that the workload has already been submitted, so it receives all the interactions of an operation, and for each work all the parameter will be injected with all the attacks. The result is a set of works with a size: workload size X Number of parameter X number of attacks. This is why we need to control the workload size. Even knowing that code coverage is very important, one should have in mind that to maximize the coverage we need to increase the workload size but this represents an huge cost both in temporal and spatial terms.

As an example, lets imagine we have an operation with 20 String type parameters, and 200 different SQL injection attacks. If we had generated 10 works (workload size = 10) the attackload will be 10 X 20 X 200 = 40000 works. Lets imagine the service is composed with 7 operations with similar characteristics the attackload size will be a 40000 X 7 = 280000. However this attackload generator has the ability to reduce the base workload to generate the attackload based on the information about the interaction previously made using a certain work.

The **Signature-based Attackload Generator** is different of the previous. In this case the entire workload is used, the reasoning for this is that we don't assume the requests based on workload have been submitted yet, thus we cannot filter works and reduce the workload size. We can, however, start with smaller workloads to reduce the size of the problem. Nevertheless the logic of injecting attacks is similar, but now the list of attacks consists in injecting signatures on the attacks first, and then injecting the result on the parameter under attack. This is made twice: one injecting the signature on the attacks and another injecting the reverse of the signature on the attacks. Further explanation for this will be provided in the description of the detecting module that uses this attackload generator. The signature is structure is *_xy_p* and the reverse of the signature is *_yx_o* where *x* corresponds to the operation id the *y* corresponds to the parameter id and *p* means it's a signature and *o* means it's a reverse signature. To conclude in this case the size of the attackload will be more then double the size of the attackload generator described previously.

4.3.3. Monitoring Module

It was implemented a **Monitor** module that is responsible to gather messages from a **Probe** and enqueue those messages to the core module through the MonitorManager component. The specific implementation of this monitor module is very simple: it consists of a RMI server that provides an interface used by the probe to report messages. Finally, the server redirects the messages to a queue on the core module.

In order to get information needed about the services we need *Probes* that can be either deployed on the web application server, or somewhere else monitoring the communication interface between the web application server and the resources used by the services (e.g. databases). Our specific probes were implemented wrapping a jdbc driver and jaxws libraries using aspect. This wrappers can intercept calls to specific methods of the jdbc driver or jaxws, get useful data and proceed with the service execution, after this the probe lookup the remote object published by RMI server on monitor module and uses the its interface to report messages.

4.3.4. Detection Modules

The detection modules have an important roles it builds a specific technique. First of all the detection module is always composed by two components, an *Identifier* and a *Test Driver.* As one can imagine not all the techniques performs exactly the same actions in the same order, thus the Test Driver is the module in which will be defined the workflow and any synchronization needed to implement the detection technique. The component identifier is responsible to contain the methods needed to analyze the available data and classify parameters of operations as vulnerable or not.

The **IPT Detection Module** can be used in services that are only *within reach*. *Within reach* services are only accessible from a user perspective, no access to the code, binaries, interfaces to resources or application server are available. In order to this module works properly the following modules should be present and selected:

- Workload generator: Pseudo-Random Workload Generator
- Attackload Generator: Regular attackload generator
- Monitoring: No

This module compares the responses to the non-malicious interaction with the responses to the malicious interactions and based on a predefined rules classifies a parameter of the given operation as vulnerable or not.

The **Sign-WS Detection Module** implements a technique is used to perform vulnerability scans to services that are *partially under control* this means we can monitor interfaces between server and database but we had no access to code or binaries, needs the following modules to be selected:

- Workload Generator: Pseudo-Random
- Attackload Generator: SignatureAttackloadGenerator
- Monitoring: Yes

In this technique it is needed to generate a workload, however in opposition to the other techniques it does not need the workload to be submitted. This technique obtains the SQL command from the messages provided by the monitoring, removes the variable parts of the commands (inputs) and if it founds the signature injected during the attackload generation it is considered a red flag. Then, if the reverse of the signature is also found the parameter of the operation identified by the signatures is classified as vulnerable. The reasoning behind the need of the signature an its reverse is that there is a chance that that there a field in same database table with the same name as the signature, so we use the reverse to outwit that possibility. Although the possibility of existence of two fields of tables of the database with the same names of the signature and its reverse is not zero, the chances are very low. Although we are using the same monitoring as we used for RAD-WS (explained next), it is not exactly the same situation, in this technique we just need the information about the SQL command, and this can be made monitoring interfaces between services and resources, this can be achieved making use of http proxies or sniffers, however we are using the same probe but not using all the information even though it still available. This was done to avoid the implementation of a different probe, but we know that on a real world situation if we did not have access at least to the binaries of the service under test the probe would not fit and other approach should be considered.

The **RAD Detection Module** implements a technique that is intended to perform scans in *under control* service, services that we have access at least to the application server and we can have access to the binaries of the service. In order to this module works properly the following modules should be present and chosen:

- Workload generator: Pseudo-Random Workload Generator
- Attackload Generator: Regular attackload generator
- Monitoring: Yes

This module identifies hotspots and builds profiles for each of them. Hotspots are interesting line of code in the service, typically line on the service where certain jdbc driver methods have been called. To do this, the driver starts to generate the workload, then it submits the workloads, during this submission messages from the monitoring are used to identify the hotspots and to profile them, profiling hotspots consist on keeping a list of non-malicious SQL commands intercept during the workload submission phase, we call this phase the *profiling phase*. Next it generates the attackload, and the submission of the malicious request is done parameter (this needs to be synchronized, otherwise it was impossible to identify the operation and the parameter under attack), during this attack phase it gets the hotspot and the SQL command from the messages gathered from monitoring and checks if the profile of the respective hotspot contains the current command, if it doesn’t, a vulnerability is reported.

This technique should not miss existing vulnerabilities, however it can report a big number of false positives if the profiling phase was not complete. In this sense the size and the quality of the workload determines directly the effectiveness of this technique.

Chapter 5
Evaluation

The goal of this chapter is to present the tasks conducted to evaluate the software developed. Instead of trying to evaluate every facet of the software, which would require a large set of tests ranging from usability tests to performance tests, we decided to focus on the evaluation of the key innovations that the tool introduces when compared with the state of the art.

The *modularity* of the tool is what allows it the possibility of being extended to develop new techniques or improve the existing ones just by adding new modules or improved ones. However, it is also necessary that this modularity does worsen the vulnerability detection capabilities of the approaches. This way, the two properties defined as most relevant to evaluate the merits of the developed tool are its modularity and the ability to report vulnerabilities equivalent to the original prototypes implementing the vulnerability detection approaches.

On the hand, evaluating the modularity of the tool is a tricky issue, since modularity is hard to express in metrics and even harder to compare when the pieces of software do not present the exact same functionalities. This way, we decided to present a modularity analysis in which we discuss the most relevant points.

On the other hand, comparing the results of our tool with the original tools is a straightforward process in which we repeat the tests performed during the evaluation of the original prototypes and then compare objectively the results obtained. Thus, in this case the evaluation presented is an experimental procedure.

The structure of the chapter is as follows. The next section discusses the modularity of the tools. The Section 5.2 presents the comparison of the results obtained by our implementation and the results achieved by the original tools.

5.1. **Modularity Analysis**

From the definition of modules presented in Section 3.3, we are able to identify two main modularity attributes. The elements (classes, interfaces, operations) inside a component should address only one concern and are strongly connected among themselves *(High Cohesion)*. At the same time a component should not address more than one concern and it should be independent of other *(Lose Coupling)*. This means they should have few connections to other components of the system.

In order to build a modular system one should first start to identify concerns and separate them, to build software components that address each concern. Once this separation is done, one can start to hide complexity of each module behind abstractions and interfaces. Hiding complexity consists is nothing more than only make public only what is needed to access to functionality of a certain module, all the other internal operation should not be visible to other modules.

Then, we have the so-called *crosscutting concerns*, that are concerns that are necessary to implement multiple functionalities and thus are present in large parts of the software and are impossible to isolate in a single module. The way these concerns are dealt with influences greatly the resulting modularity of the software.

5.1.1. How modular is our tool?

There are a lot of metrics for degrees of modularization. However almost all of them are for modularization of an entire project, consisting in analyses of packages, classes and interfaces and counting the dependencies between these elements to compute modularity degree. n our case, we just want to assure that certain concerns are modular, for instance, the GUI of the tool and other concerns related with generic functionalities were not treated the same way as concerns related specific functionalities, i.e. with the vulnerability detection approach.

Another problem with this metrics is the lack of meaning of the values it produces in absolute terms. Like many other metrics, we always need to compare two systems, to provide meaning to the values. This means we cannot determine how modular an implementation. Instead, it is possible to say that one implementation is more modular than other, but even this comparison will only make sense if we are comparing different implementations of the exactly same functionalities. Finally, it is not a fair assessment to say that a program implementation is more modular than other if the goals and functionalities implemented are different because concerns are different and the number of crosscutting concerns may be higher from one implementation to the other.

Due to these limitations and since comparison between our integrated tool and the existing ad-hoc tools would not be fair, we decided to perform a qualitative analyses instead of a quantitative analyses. To do that, we analyzed our component diagram in order to identify design patterns that can give us evidence that the components are compliant with modularity attributes.

As we already knew we want to use modularization on our project, we considered it from the beginning and our architecture was defined taking the modular attributes into account. We defined the design rules that guided the definition of the module components. These rules are:

- An external component *does not interact* with other external component;
- A component shall never provide functionality that is not related with the concerned that it addresses;
- An external component *provides an interface* to the core module;
- An external component *may use an interface* provided by the core module;
- In the core module there is a component to interact with the external component that addresses a certain concern;
- External components should only interact with the respective internal component.
- Internal components can interact with each other in other to achieve the complete tool goal.

The main goal of our tools is to help on improvement and implementation of new web services vulnerability detection techniques based on testing. Instead of concentrating only on identification of concerns of the entire system we focused on the concerns related to the techniques themselves. This means that our interest was to modularize the key aspects of a vulnerability detecting technique. As shown on section 3.3.1. , we identified four concerns that are necessary to be possible to develop, change or maintain without knowing or change the rest the system. Those concerns are: workload generation, attackload generation, monitoring and detecting.

The concerns mentioned above are particularly relevant since they are the dynamic aspects of a vulnerability detection technique. Thus, we decided to modularize them in order to easily change, replace or simply add new ones and use them with our core application. This option will allow us to easily improve and implement new techniques and at the same time have them available in one single tool, giving us the ability to use the technique that best fits each specific service. The other concerns are considered crosscutting concerns as they are common among all the techniques and they are all elements of the core component.

5.1.2. Modularity qualitative assessment

As an example of modularity, we will discuss the modular attributes of both detecting and monitoring external modules. First, if we check the concern that each component of the partially component diagram represent in **Figure 11** we have four components, the external monitor and the MonitorManager component that are assign to the monitoring concern and external detecting module and DetectingManager that deals with the detecting concern.

As it is possible to observe, the external components have few connectivity (one) and that connectivity is through and interface to a component that addresses exactly the same concern. Also, this component does not depend on others. These are typical characteristics of a modular design, and the less connection to other system component and the less concerns are addressed by one component the more modular is the component under analyzes. Finally, high cohesion can be observed in the external modules because, more specifically functional cohesion, functional cohesion is when all parts of the module are together because they share the same goal, this is all the elements present on the module address the same concern, and all of them contribute to the same task.

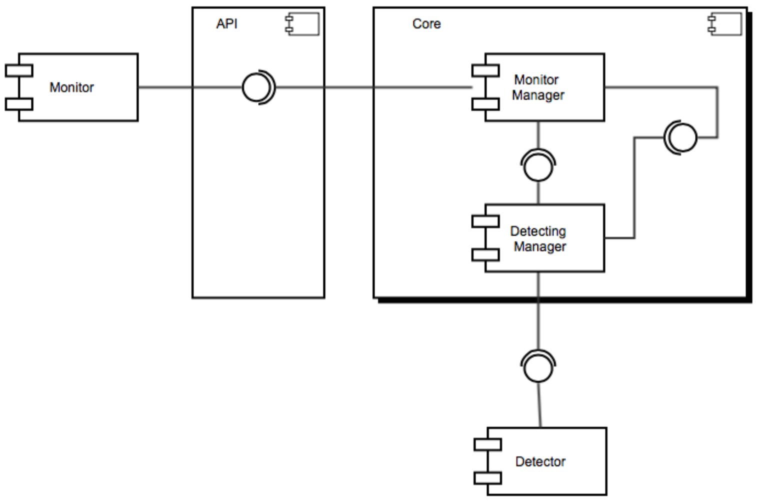


Figure 11 – Component Diagram for Monitoring and Detecting

All of the external modules are very similar, that way we took only this few components to illustrate the modular analyses, another reason to choose these specific one was also to demonstrate that there are also drawbacks in a modular design. As mentioned before in this document the message from the external monitor should reach the external detecting manager, since following the modular design guideline we should avoid interaction or relation of dependencies between modules, and we should avoid them even more if they are addressing different concerns. Thus, it is easy to understand that the travel of message from the external monitor to the external detecting module is way more complex in a modular approach then on a non-modular approach. The message is pushed by the external monitor to the monitor manager message queue, the monitor manager calls a function from the detecting manager which in turn calls a method from the interface provided by the external detecting to put the message on a message queue on the external detecting. Also not every external detection modules requires the same steps in the same order, because of that the test driver is placed in the detection module, so when the external detection module needs some action to be performed it will request to the detecting manager that will ask to the respective internal module that will ask to the respective external module.

Another problem we faced with the modular design approach was the synchronization, as one can imagine in some techniques the messages receives must be synchronized with the service and operation being tested in that moment. This pitfalls my have an impact on the system performance, and in some cases it increases the complexity of certain parts of the system making them harder to maintain, however that parts are inside the core module that we expect to need a few modification on the future, but on other hand keep the external modules high modular, that are the ones that more will be added, removed, or suffer modifications.

5.2. **Comparison with Original Tools**

In order to test if the implementation of different techniques is possible using the proposed modular approach, we implemented the necessary modules to emulate three techniques that are implemented ad-hoc. We have the detailed results for the three different tool implemented in [40]. The methodology used consisted in the implementation of the necessary modules to complete their techniques, replicate the scenario described by the authors, and compare the results in terms of vulnerability detection rate and false positives. If the results are similar we can say that the approach followed is valid, and that we can integrate various techniques, implementing external modules and combining them.

5.2.1. Experimental Setup

Four machines compose our experimental setup, a machine with the soascanner installation, two machines with JBoss application servers and a fourth machine with Oracle database installation. The two machines with the JBoss application server installation, one represents the services with in reach and the other is used to simulate the partially under control and under control services, both machines use the same DBMS. All the machines are virtual machines managed by the virtualization system XenServer 6.0 installed in a physical machine.

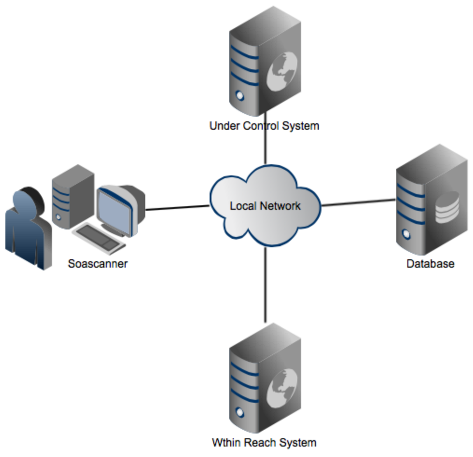


Figure 12 – Technical setup for the experiments

In order to validate our approach we decided to submit our tool and external modules implementation to the benchmark for web services vulnerability scanners proposed in [41]. This benchmark consists in a set of web services at are an adaption of three standard benchmarks developed by the Transactions processing Performance Council, namely: TPC-App, TPC-C, and TPC-W [42] TPC-App is a performance benchmark for web services infrastructures and specifies a set of web services accepted as representative of real environments. TPC-C is a performance benchmark for transactional systems and specifies a set of transactions that include entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. Finally, TPC-W is a benchmark for web-based transactional systems. The business represented by TPC-W is a retail store over the Internet where several clients access the website to browse, search, and process orders.

Rather then introducing artificial SQL vulnerabilities injection, they asked to an external developer to implement the TPC-app services (without disclosing the propose) and asked the same external programmer to implement TPC-C and TPC-W to the form of web services. This approach turns the workload of the benchmark more realistic because any vulnerability found on those services were inadvertently introduced by the programmer, which what happens most of the time in real world conditions. They had invited three external developers with two or more years of experience on security in database centric applications, who formally inspected the code looking for vulnerabilities, and as expected they had found several vulnerabilities in the web services implementation. They were asked to point both the vulnerable parameters and the source code lines prone to SQL injection attacks.

With this information the authors created several version of teach web services: a version without vulnerabilities, N version with one vulnerability, a one versions with N vulnerabilities, and of course the original version, with all the vulnerabilities.

We had access to the implementation of this benchmark and we had installed is as shown in picture **Figure 12**, however we installed in two different machines, one machine is under control, that means that we can deploy a probe, and the other in with in reach, it means we have no access to the services source code nor to the interfaces, thus no monitoring is possible.

We have a machine client where our application soascanner is running, and we have a database we provided by the author of the benchmark.

The old tools have been submitted to this benchmark also, the intention in to submit our tool and see if the results are similar, if yes, its proves that we can build techniques combining external modules to implement different techniques and have all of hem integrated at the same tool.

5.2.1. Benchmark results

To compare the both modular approach and the ad-hoc tools, have had run the benchmark described in the previous section to our tool, and we have the results obtained for the ad-hoc tool kindly provided by the author of the tools and the proposed benchmark. Since our goal is to prove that we can implement the same techniques as the ad-hoc tools, we have just used to metrics, *coverage* and *false positive rate.* The metric *coverage* gives us the ratio between reported vulnerabilities and the total known vulnerabilities, and the *false positive rate* is the number of false positives by the number of reported vulnerabilities. We had calculated this metrics for the three tools in three different scenarios, when scanning version of the services without known vulnerabilities, version of services with some of the parameters vulnerable, and version of services with all the vulnerabilities known.

After the analyses of the results of the ad-hoc tools, we can notice that IPT-WS is the one that presents worst result, in average it has a low coverage, and at the same time it reports a big percentage of false positives, as expected the performance of this technique is limited due to lack of information compared with the other two techniques. IPT-WS compares responses to valid requests with reposes to the attacks, which makes the detection of vulnerable parameters a hard task. On other hand the other two techniques perform quiet well, on one hand they do not report false positives and on the other they show a high coverage. However its interesting the fact that both techniques perform better in the scenario that contains the versions of the services with all the known vulnerabilities than when scanning version of the services that only has some of the parameter vulnerable.

	REVIEW	T.P.	REPORTED	F.P.	COVERAGE	F.P. RATE
IPT-WS	158	38	67	29	**24%**	**43%**
RAD-WS		119	119	0	**75%**	**0%**
Sign-WS		117	117	0	**74%**	**0%**

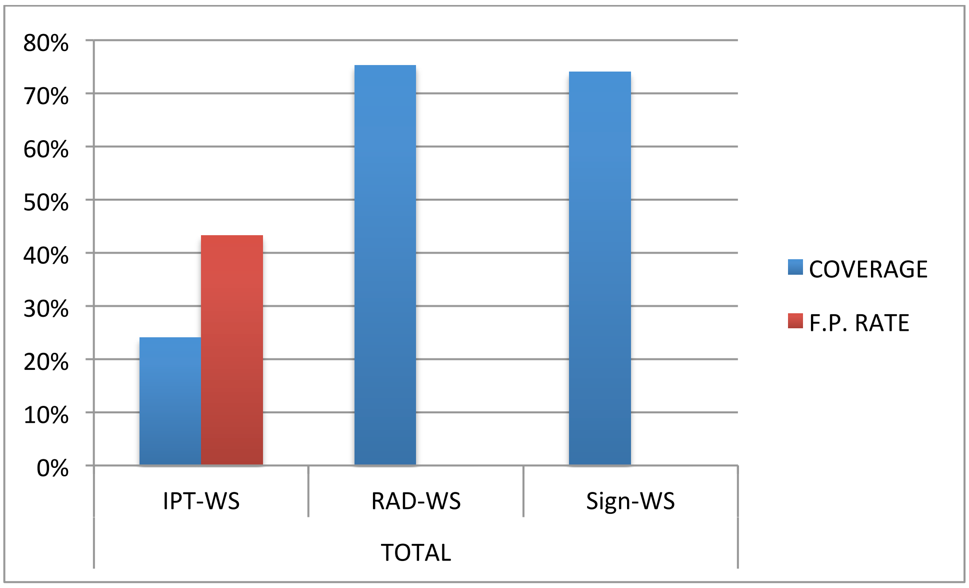


Figure 13 - Ad-hoc tools benchmark results

Analyzing the results of our modular integrated tool, we can observe that the results are very similar when it comes to coverage, all the techniques on the different scenarios present identical results, varying between 1-2%, both RAD-WS and Sign-WS do not present false positives as well. Another similarity is the fact that this techniques to perform better in the scenario that contains the versions of the services with all the known vulnerabilities than when scanning version of the services that only has some of the parameter vulnerable. However in the case of the false positive rate for IPT-WS is significantly lower, it reports almost half of the false positive then the ad-hoc tool. This can be explained by the fact that the source code of the version of the IPT-WS had already some improvements that we were not able to use in the evaluation of its standalone version. The little deviations on the result for RAD-WS and Sign-WS were expected since there is little a non-determinism on this techniques, the attack are based on random workloads, it is possible to some times reach some lines of code and other do not. In the particular case of RAD-WS implies a learning phase, things like the workload size may interfere with results.

	REVIEW	T.P.	REPORTED	F.P.	COVERAGE	F.P. RATE
M(IPT-WS)	158	38	50	12	**24%**	**24%**
M(RAD-WS)		117	117	0	**74%**	**0%**
M(Sign-WS)		118	118	0	**75%**	**0%**

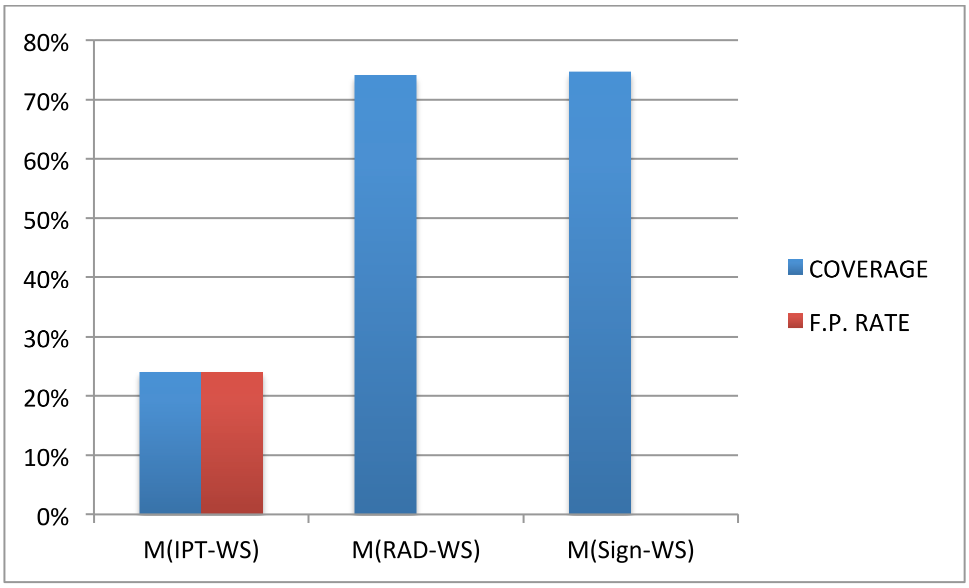


Figure 14 –Integrated Tool Benchmark results

Chapter 6
Tool Manual

As referred previously, we consider 3 main classes of users for this tool: **service consumers, service providers** and, **advanced users**. Although the differences between consumers and is limited to the access they have to the services to be tested, the differences from both to the advanced users are significant. In fact, besides using the tool to test service infrastructure, advanced users have the possibility to **extend the tool** by developing new modules that will improve existing vulnerability detection techniques or even allow creating new ones. This way, more than just presenting a common usage manual, it is also necessary to explain how the advanced users can extend the tool.

The goal chapter contains the manuals of the tool for both testers (consumers and providers) and for developers (advanced users). For developers it is provided a list of guidelines and good practices to successfully implement a new module. Additionally, it will be briefly explained how to configure a new a vulnerability detection technique. As for all testers, we will show how to configure the tool and perform a vulnerability testing campaigns, taking maximum advantage of the functionalities of the tool. This is mainly a graphical task, thus the user will be guided through the tool's graphical user interface.

The structure of the chapter is as follows. The next section presents the user guide for developers while the Section 6.2 presents the user guide for testers.

6.1. **Developer User Guide**

6.1.1. Building a new module

Building a new external module is in general a straightforward task. However there are some details and some constraints the one must be aware of when the implementation of the external modules.

An external module is nothing more than a java archive file, a class or set of classes, where exists at least one implementation of a class that inherits one of those abstract classes: AbstractWLGenerator, AbstractALGenerator, AbstractMonitoring or AbstractDetecting.

The main dependency necessary to build the modules will also be necessary is the soascanner-api.jar library. This library includes the classes necessary to compile the external module and that contains the necessary interfaces of the core module. Implementation of auxiliary classes is allowed, but the constructor of the specific implementations of the abstract classes shall be empty, because these classes are instantiated in runtime and the core has no knowledge about constructor parameter.

6.1.2. Create a new vulnerability detection technique

The task of creating and configuring a new vulnerability detection technique is mainly dependent on the implementation of a new external discovery module. This implementation follows the same steps defined in the previous subsection. However, when implementing an external discovery module, the developer needs to have in mind that a new *TestDriver* class should be implemented. Although this is not strictly required, it is highly recommended because this class is in charge of configuring and staring the execution of the remaining modules.

Finally, it is very often necessary to implement threads handle the messages from the message queue of the external module. The methods existent in the *DetectingManagerInterface* instance in the *AbstractDetecting* class can be used to drive the execution of a detection technique but concerns like synchronization and order of the calls are of the programmer responsibility.

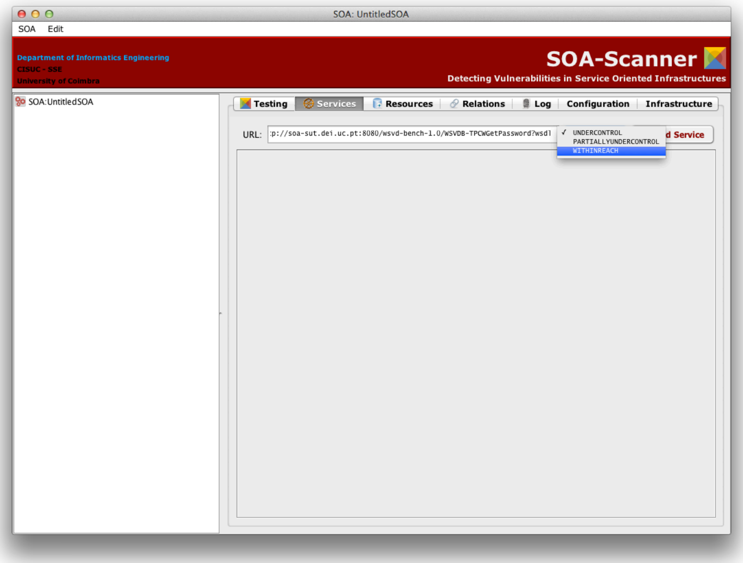
6.2. **Tester User guide**

This section describes how the tools can be used from a test carrier perspective, we will focus on the steps needed to configure the tool in order to perform vulnerability scan to a set of services.

6.2.1. Adding Services

In order to add services to the infrastructure to test:

- Go to the tab *Services*
- Insert the web service wsdl file url
- Choose the service type
- Press button *Add Service* (see **Figure 15**)



URL: :p://soa-sut.dei.uc.pt:8080/wsvd-bench-1.0/WSVDB-TPCWGetPassword?wsdl

✓ UNDERCONTROL
PARTIALLYUNDERCONTROL
WITHINREACH

Figure 15 – Adding a service to the infrastructure

As shown in **Figure 16**, if the service has ben successfully imported it can be seen the web service listed bellow, with a green tick in front of him, a user can insert as many services as wanted. Also the service should now be listed in the tree list on the left.

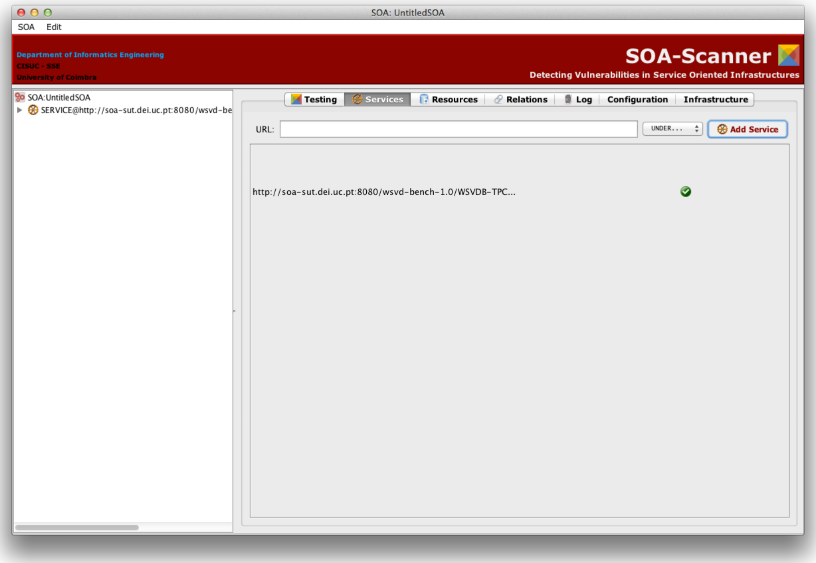

SOA: UntitledSOA
SOA Edit
Department of Informatics Engineering
CISUC - SSE
University of Coimbra
SOA-Scanner
Detecting Vulnerabilities in Service Oriented Infrastructures
SOA:UntitledSOA
SERVICE@http://soa-sut.dei.uc.pt:8080/wsvd-be
Testing
Services
Resources
Relations
Log
Configuration
Infrastructure
URL:
UNDER...
Add Service
http://soa-sut.dei.uc.pt:8080/wsvd-bench-1.0/WSVDB-TPC...

Figure 16 – Service successfully added outcome.

Also there is available a visual infrastructure graph representation, it is updated every time a resource is added to the infrastructures, either manually or automatically. To see it just navigate to the *Infrastructure* tab. **Figure 17** is the state of the infrastructure after the user inserted a service manually.

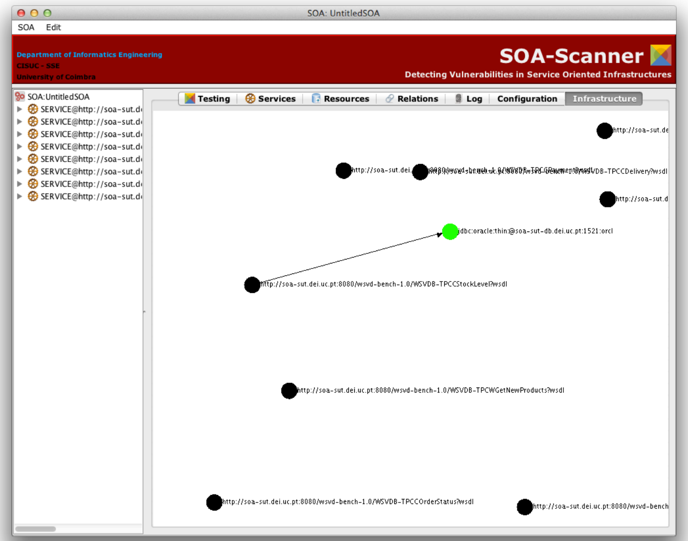

SOA: UntitledSOA
SOA Edit
Department of Informatics Engineering
CISUC - SSE
University of Coimbra
SOA-Scanner
Detecting Vulnerabilities in Service Oriented Infrastructures
SOA:UntitledSOA
SERVICE@http://soa-sut.d
Testing
Services
Resources
Relations
Log
Configuration
Infrastructure
http://soa-sut.de
http://soa-sut.dei.uc.pt:8080/wsvd-bench-1.0/WSVDB-TPCCDelivery?wsdl
http://soa-sut.d
jdbc:oracle:thin:@soa-sut-db.dei.uc.pt:1521:orcl
http://soa-sut.dei.uc.pt:8080/wsvd-bench-1.0/WSVDB-TPCCStockLevel?wsdl
http://soa-sut.dei.uc.pt:8080/wsvd-bench-1.0/WSVDB-TPCWGetNewProducts?wsdl
http://soa-sut.dei.uc.pt:8080/wsvd-bench-1.0/WSVDB-TPCCOrderStatus?wsdl
http://soa-sut.dei.uc.pt:8080/wsvd-bench

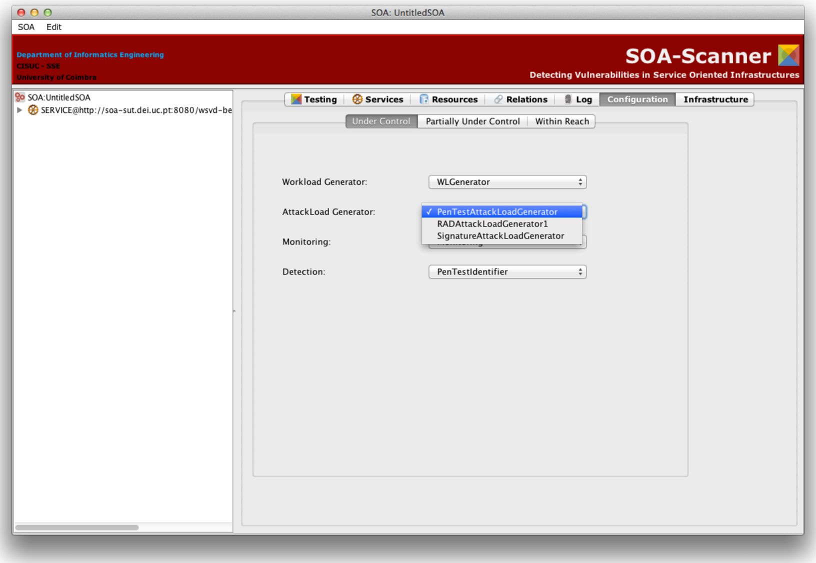
Figure 17 – Infrastructure graph visual representation

6.2.2. Configure for scanning

Before start scanning services, the user should configure which modules to use in order to build the technique that will perform a scan for each service type (**Figure 18**).

- To configure modules before scanning:

- Navigate to *Configuration* tab
- For each service type tab: *Under control*, *Partially Under Control* and *Within Reach* select from the drop down menus a module for *workload generator*, *attackload generator*, *monitoring* and *detection*.



Workload Generator: WLGenerator

AttackLoad Generator: ✓ PenTestAttackLoadGenerator / RADAttackLoadGenerator1 / SignatureAttackLoadGenerator

Monitoring:

Detection: PenTestIdentifier

Figure 18 – Configuring modules per service type

6.2.3. Preparing service to scan

Before starting a scan, some edit to the parameter attributes of the services may be needed, have in mind that is possible that the workload generator may use this information to generate better workloads. To do so:

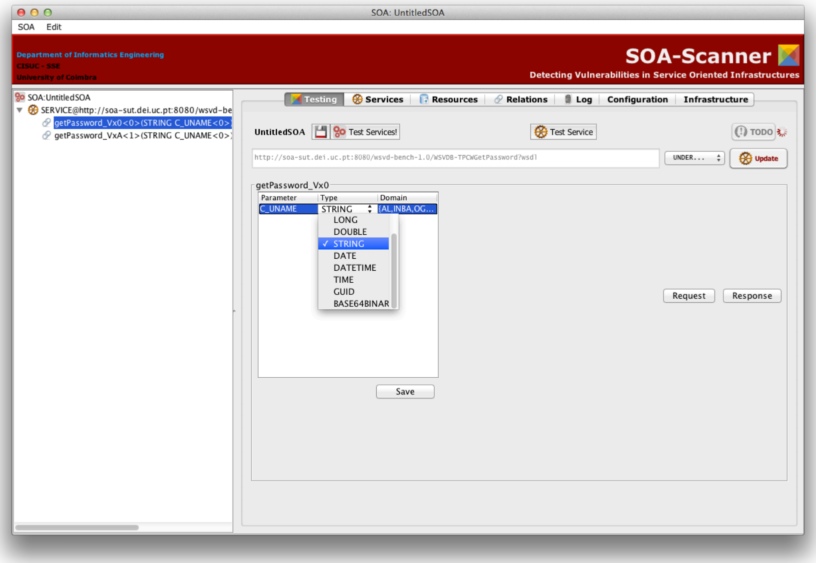
- Navigate to *Testing* tab
- Double click a service on the tree list on the left
- Select the operation and the parameter will be listed
- Change the parameter data type as shown in the **Figure 19**
- Change the domain editing the string inside the domain field of the list

It is also possible to change the service type in this tab to do that just change the type and press *Update* on the right.

There is a special syntax to define a domain:

- Enumeration
  {a,b,c} one of the value a,b or c
- Length
  |a<->b| any size from a to b(strings)
- Bound
  [a-b] any value from a to b

Figure 19 depicts how these changes can be done.



SOA:UntitledSOA
SERVICE@http://soa-sut.dei.uc.pt:8080/wsvd-be
getPassword_Vx0<0>(STRING C_UNAME<0>)
getPassword_VxA<1>(STRING C_UNAME<0>)

http://soa-sut.dei.uc.pt:8080/wsvd-bench-1.0/WSVDB-TPCWGetPassword?wsdl

getPassword_Vx0

Parameter	Type	Domain
C_UNAME	STRING	{AL,INBA,OG...

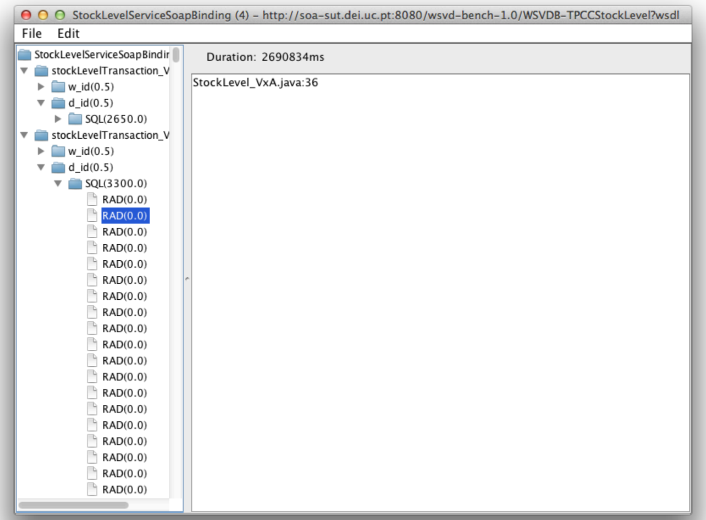
LONG
DOUBLE
[✓ STRING]
DATE
DATETIME
TIME
GUID
BASE64BINAR

Figure 19 – Changing parameters data type and domain

6.2.4. Performing a scan

To start scanning:

- Go to the tab *Testing*
- Select a service and press the button *Test Service* or just press *Test Services* to test them all
- Wait for the Report Viewer GUI to appear (**Figure 20**)


StockLevelServiceSoapBinding (4) – http://soa-sut.dei.uc.pt:8080/wsvd-bench-1.0/WSVDB-TPCCStockLevel?wsdl
File
Edit

```
StockLevelServiceSoapBindir
▼ stockLevelTransaction_V
  ▶ w_id(0.5)
  ▼ d_id(0.5)
    ▶ SQL(2650.0)
▼ stockLevelTransaction_V
  ▶ w_id(0.5)
  ▼ d_id(0.5)
    ▼ SQL(3300.0)
        RAD(0.0)
        RAD(0.0)
        RAD(0.0)
        RAD(0.0)
        RAD(0.0)
        RAD(0.0)
        RAD(0.0)
        RAD(0.0)
        RAD(0.0)
        RAD(0.0)
        RAD(0.0)
        RAD(0.0)
        RAD(0.0)
        RAD(0.0)
        RAD(0.0)
        RAD(0.0)
        RAD(0.0)
        RAD(0.0)
        RAD(0.0)
```

```
Duration: 2690834ms
StockLevel_VxA.java:36
```

Figure 20 – Report Viewer GUI

During scans the list of resources will be updated when a new resource is found by the monitoring (if there is one), at the same time the infrastructure graph is also updated check **Figure 21**.

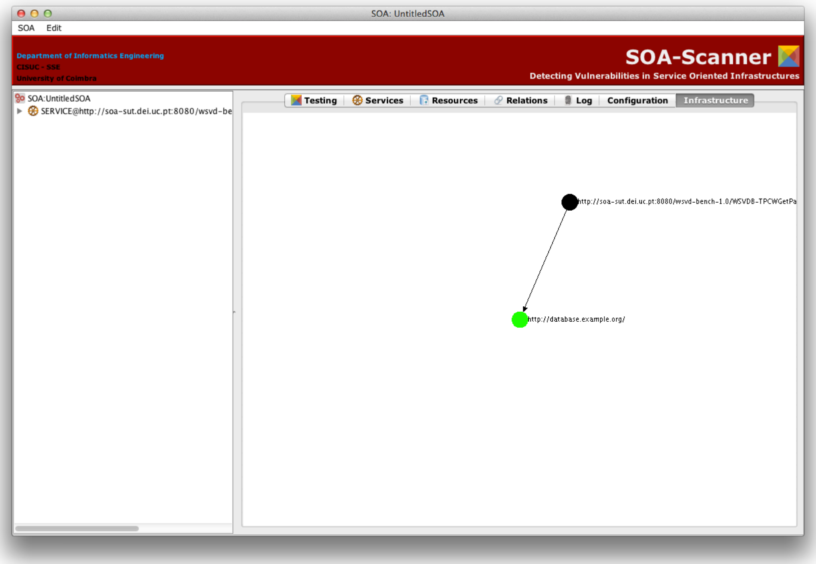

SOA: UntitledSOA
SOA
Edit
Department of Informatics Engineering
CISUC - SSE
University of Coimbra
SOA-Scanner
Detecting Vulnerabilities in Service Oriented Infrastructures
SOA:UntitledSOA
SERVICE@http://soa-sut.dei.uc.pt:8080/wsvd-be
Testing
Services
Resources
Relations
Log
Configuration
Infrastructure
http://soa-sut.dei.uc.pt:8080/wsvd-bench-1.0/WSVDB-TPCWGetPa
http://database.example.org/

Figure 21 – Database found during the scan

When the scan finishes will presented windows with reports, one for each service tested (see **Figure 20**), the information contained on the reports varies with external detecting module.

6.2.5. Adding New Resources and Relations

In some cases it is not possible to discover new resources automatically, so it is possible to add new Resources and Relations.

To add a Resource:

- Navigate to *Resources* tab
- Insert the Resource url
- Choose the resource type on the left
- Press button Add resource

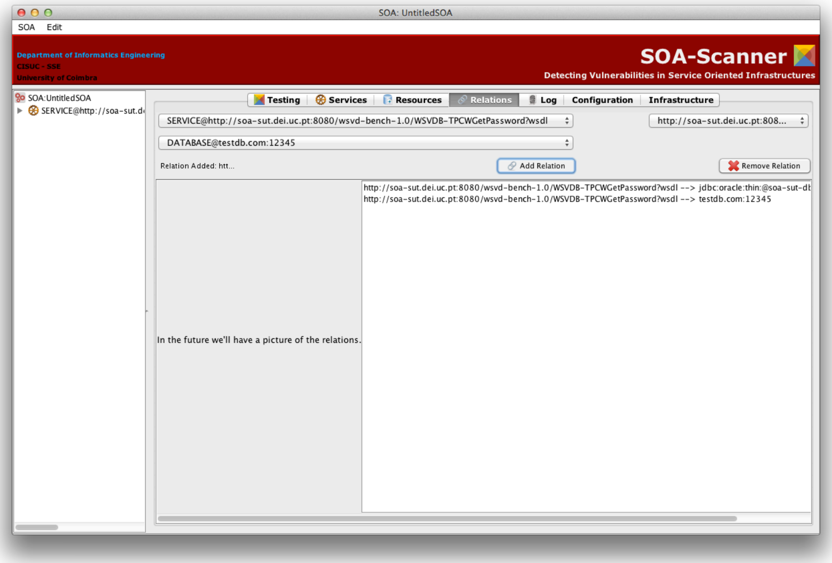
After adding the resource the infrastructure the graph will be updated (only for databases in the current version of the tool);

To add a relation (**Figure 22**):

- Go to the tab relation
- Go to the drop down menu on the top left and choose a source resource (only service will be listed)
- Go to the drop down menu bellow the mentioned before and choose a target resource.
- Press *Add Relation*

To remove a relation:

Choose the relation to remove from the list on the right and press *Remove Relation button*



SERVICE@http://soa-sut.dei.uc.pt:8080/wsvd-bench-1.0/WSVDB-TPCWGetPassword?wsdl

http://soa-sut.dei.uc.pt:808...

DATABASE@testdb.com:12345

Relation Added: htt...

In the future we'll have a picture of the relations.

```
http://soa-sut.dei.uc.pt:8080/wsvd-bench-1.0/WSVDB-TPCWGetPassword?wsdl --> jdbc:oracle:thin:@soa-sut-dt
http://soa-sut.dei.uc.pt:8080/wsvd-bench-1.0/WSVDB-TPCWGetPassword?wsdl --> testdb.com:12345
```

Figure 22 – Resources Management.

Chapter 7
Conclusion

This work proposed an integrated tool based on a modular design that aims to help the evolution of web service vulnerability detection techniques. The implemented tool has the particularity of dividing the implementation of vulnerability detection techniques in simple and independent modules, making them easy to understand and evolve and maintain.

As more modules are developed and added to this new integrated tool, more techniques are available at the same application. Also, it as the added value to enable the test of the several services present in a service-based infrastructures depending on the access level one has to such services.

To validate the concepts of modular and integrated tool we provided the implementation of some known techniques, and we submitted our tool to the a benchmark that the ad-hoc existing tools had also been subjected, and the results shown that those techniques can be implemented using the modular tool, showing in most of the case similar or better results in term of coverage and false positive rate. This way, we consider that most important goals of this work have been achieve.

Future work includes publishing the tool to be used by interested testers and developers. We believe that it has the potential to open a window for future research as the existence of such modular tool can incentive the entire research community to collaborate with the implementation of new and improved modules, thus improving the state of the art in web services security assessment.

References

[1] "What Is SOA?" [Online]. Available: http://www.opengroup.org/soa/source-book/soa/soa.htm#soa_definition. [Accessed: 24-Jan-2014].

[2] K. Bennett, P. Layzell, D. Budgen, P. Brereton, L. Macaulay, and M. Munro, "Service-based software: the future for flexible software," in *Software Engineering Conference, 2000. APSEC 2000. Proceedings. Seventh Asia-Pacific*, 2000, pp. 214–221.

[3] N. Antunes and M. Vieira, "Defending against Web Application Vulnerabilities," *Computer*, vol. 45, no. 2, pp. 66–72, 2012.

[4] "Top 10 2013-Top 10 - OWASP." [Online]. Available: https://www.owasp.org/index.php/Top_10_2013-Top_10. [Accessed: 26-Jan-2014].

[5] M. Vieira, N. Antunes, and H. Madeira, "Using web security scanners to detect vulnerabilities in web services," in *IEEE/IFIP International Conference on Dependable Systems Networks, 2009. DSN '09*, 2009, pp. 566–571.

[6] "Injection Flaws - OWASP." [Online]. Available: https://www.owasp.org/index.php/Injection_Flaws. [Accessed: 28-Jan-2014].

[7] A. Singhal, T. Winograd, and K. Scarfone, "Guide to secure web services," *NIST Spec. Publ.*, vol. 800, no. 95, p. 4, 2007.

[8] "OWASP Application Security FAQ - OWASP." [Online]. Available: https://www.owasp.org/index.php/OWASP_Application_Security_FAQ#What_are_application_firewalls.3F_How_good_are_they_really.3F. [Accessed: 02-Sep-2014].

[9] "SQL Injection - OWASP." [Online]. Available: https://www.owasp.org/index.php/SQL_Injection. [Accessed: 28-Jan-2014].

[10] "XPATH Injection - OWASP." [Online]. Available: https://www.owasp.org/index.php/XPATH_Injection. [Accessed: 28-Jan-2014].

[11] "FindBugs^{TM} - Find Bugs in Java Programs." [Online]. Available: http://findbugs.sourceforge.net/. [Accessed: 20-Dec-2013].

[12] "IntelliJ IDEA :: Static Code Analysis." [Online]. Available: http://www.jetbrains.com/idea/documentation/static_code_analysis.html. [Accessed: 19-Dec-2013].

[13] "Flawfinder Home Page." [Online]. Available: http://www.dwheeler.com/flawfinder/. [Accessed: 20-Dec-2013].

[14] "WSDigger | McAfee Free Tools." [Online]. Available: http://www.mcafee.com/us/downloads/free-tools/wsdigger.aspx. [Accessed: 15-Jan-2014].

[15] "WSFuzzer | Free Development software downloads at SourceForge.net." [Online]. Available: http://sourceforge.net/projects/wsfuzzer/. [Accessed: 26-Jan-2014].

[16]“Penetration Testing, Web Application Security | HP® Official Site.” [Online]. Available: http://www8.hp.com/us/en/software-solutions/software.html?compURI=1341991#.UrRgdXm60b4. [Accessed: 20-Dec-2013].

[17]“IBM - Software - IBM Security AppScan - United States.” [Online]. Available: http://www-03.ibm.com/software/products/en/appscan. [Accessed: 20-Dec-2013].

[18]“Website Security with Acunetix Web Vulnerability Scanner | Web Vulnerability Scanner.” [Online]. Available: http://www.acunetix.com/. [Accessed: 20-Dec-2013].

[19]W. G. J. Halfond and A. Orso, “AMNESIA: Analysis and Monitoring for NEutralizing SQL-injection Attacks,” in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, New York, NY, USA, 2005, pp. 174–183.

[20]“The Benefits of Acunetix WVS AcuSensor | Acunetix.” [Online]. Available: http://www.acunetix.com/web-security-articles/rightwvs/. [Accessed: 26-Jan-2014].

[21]N. Antunes and M. Vieira, “Benchmarking Vulnerability Detection Tools for Web Services,” in *2010 IEEE International Conference on Web Services (ICWS)*, 2010, pp. 203–210.

[22]N. Antunes and M. Vieira, “Detecting SQL Injection Vulnerabilities in Web Services,” in *Fourth Latin-American Symposium on Dependable Computing, 2009. LADC ’09*, 2009, pp. 17–24.

[23]N. Antunes and M. Vieira, “Enhancing Penetration Testing with Attack Signatures and Interface Monitoring for the Detection of Injection Vulnerabilities in Web Services,” in *2011 IEEE International Conference on Services Computing (SCC)*, 2011, pp. 104–111.

[24]N. Antunes, N. Laranjeiro, M. Vieira, and H. Madeira, “Effective Detection of SQL/XPath Injection Vulnerabilities in Web Services,” in *IEEE International Conference on Services Computing, 2009. SCC ’09*, 2009, pp. 260–267.

[25]D. A. Chappell and T. Jewell, *Java Web services*. Sebastopol, CA: O’Reilly, 2002.

[26]“WSDL and UDDI.” [Online]. Available: http://www.w3schools.com/webservices/ws_wsdl_uddi.asp. [Accessed: 08-Jan-2014].

[27]L. Richardson and S. Ruby, *RESTful web services*. O’Reilly, 2008.

[28]“WhiteHat Security Reveals New Trends in Web Vulnerabilities with Annual Website Security Statistics Report.” [Online]. Available: https://www.whitehatsec.com/resource/stats.html. [Accessed: 26-Jan-2014].

[29]“Static Code Analysis in the NetBeans IDE Java Editor.” [Online]. Available: https://netbeans.org/kb/docs/java/code-inspect.html. [Accessed: 27-Jan-2014].

[30]N. Laranjeiro, M. Vieira, and H. Madeira, “Improving Web Services Robustness,” in *IEEE International Conference on Web Services, 2009. ICWS 2009*, 2009, pp. 397–404.

[31]J. Edvardsson, “A survey on automatic test data generation,” in *Proceedings of the 2nd Conference on Computer Science and Engineering*, 1999, pp. 21–28.

[32]V. Santiago, A. S. M. do Amaral, N. L. Vijaykumar, M. F. Mattiello-Francisco, E. Martins, and O. C. Lopes, "A Practical Approach for Automated Test Case Generation using Statecharts," in *Computer Software and Applications Conference, 2006. COMPSAC'06. 30th Annual International*, 2006, vol. 2, pp. 183–188.

[33]M. De Barros, J. Shiau, C. Shang, K. Gidewall, H. Shi, and J. Forsmann, "Web Services Wind Tunnel: On Performance Testing Large-Scale Stateful Web Services," in *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2007. DSN '07*, 2007, pp. 612–617.

[34]L. Koskela, "JavaRanch Journal - January 2004 - Introduction to Code Coverage." [Online]. Available: http://www.javaranch.com/journal/2004/01/IntroToCodeCoverage.html. [Accessed: 16-Dec-2013].

[35]L. Baresi and S. Guinea, "Towards Dynamic Monitoring of WS-BPEL Processes," in *Service-Oriented Computing - ICSOC 2005*, B. Benatallah, F. Casati, and P. Traverso, Eds. Springer Berlin Heidelberg, 2005, pp. 269–282.

[36]J. Irwin, G. Kickzales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, and J. Loingtier, "Aspect-oriented programming," *Proc. ECOOP IEEE Finl.*, pp. 220–242, 1997.

[37]R. Ventura, M. Vieira, and N. Antunes, "Software Requirements Specification for An Integrated Tool to Detect Vulnerabilities in Service-Based Infrastructures." [Online]. Available: http://student.dei.uc.pt/~ventura/SoftwareRequirementsSpecification.pdf.

[38]C. Y. Baldwin and K. B. Clark, *Design rules: The power of modularity*, vol. 1. MIT press, 2000.

[39]W. L. Hürsch and C. V. Lopes, "Separation of Concerns," 1995.

[40]N. Antunes and M. Vieira, "Detecting SQL Injection Vulnerabilities in Web Services," in *Fourth Latin-American Symposium on Dependable Computing 2009 (LADC '09)*, Joao Pessoa, Brazil, 2009, pp. 17–24.

[41]N. Antunes and M. Vieira, "Benchmarking Vulnerability Detection Tools for Web Services," in *IEEE Eighth International Conference on Web Services (ICWS 2010)*, Miami, Florida, USA, 2010, pp. 203–210.

[42]"TPC - Benchmarks." [Online]. Available: http://www.tpc.org/information/benchmarks.asp. [Accessed: 02-Sep-2014].