Emulation of Malformed XML Using WSInject for Security Testing Against WS-Security

Marcelo Palma Salas, Eliane Martins
Institute of Computing, State University of Campinas
Campinas, Brazil
Email: {marcelopalma,eliane}@ic.unicamp.br

Abstract— **Web Services is a technology that provide more connectivity, flexibility and interoperability among their applications. Due to its distributed and open nature, it is susceptible to Malformed XML attack, which inserts malicious code in the SOAP message request. This attack causes errors in the XPath parser in order to generate server failures (crash) that expose confidential information as part of the response. One countermeasure is to employ Security Testing, which allows the detection of this type of vulnerability and helps to discover new ones, before they are exploited by attackers. Our goal is to use the WSInject fault injector, which emulates Malformed XML attack for Security Testing against WS-Security with Security Tokens, which ensures authentication and authorization of the messages exchanged. The results were compared with a Vulnerability Scanner, which reproduces this type of attack, getting better results with WSInject.**

Keywords - Malformed XML; fault injector; WSInject; Security Testing; WS-Security; Web Services Security.

I. INTRODUCTION

Using the Web Service technology, an application can invoke another one to make simple or complex tasks even if the two applications are in different systems and written in different languages, making their resources available to any client that can operate them. However, new security challenges are created, since they are open and distributed.

There are many studies [1-5] about the various attacks on Web Services and their protocols such as SOAP and XML, like injection attacks, for example, which were the most exploited in 2010, according to the Open Web Application Security Project (OWASP Top Ten). These attacks are composed by XML Injection, SQL Injection, XPath Injection, Cross-site Scripting (XSS), Malformed XML, among others. Injection attacks occur when user-supplied data are sent to a parser as part of a command or query. In this way, the parser is fooled by the request and run malicious commands or manipulate data, generating errors in the target system.

The protection against such attacks requires security mechanisms applied to the SOAP message to ensure its transportation. In April 2004, WS-Security specification was published by OASIS, introducing a large number of new specifications and technologies to protect the communication of SOAP messages against several types of attacks such as those mentioned above.

Our goal is to use a Fault Injector (FI) to test security in the messages exchange among clients and Web Services, rather than using other tools like Vulnerabilities Scanners (VS), in order to obtain: (1) more coverage of attacks; and (2) lower number of false positive. With respect to (1), the use of an injector allows to emulate different types of attacks, varying the parameters and data injected. In (2), we used a set of rules, based on various sources.

The experiments are organized in two phases. In the pre-analysis phase the Vulnerability Scanner (VS) soapUI is applied to ten real Web Services, in order to obtain a data set, being that five of them use WS-Security and five does not. This information, based on results from other researches, helped us to develop a set of rules to determine whether an attack was successful. In the second phase, we applied the fault injector WSInject in the same Web Services. This injector emulates attacks and injects faults to the Web Services. In both phases they were tested in the presence of Malformed XML, which is ideal for analyzing the robustness of Web Services. These attacks insert malformed XML fragments, leaving tags open, adding attributes not defined, among others, causing possible failures (crash) in Web Services [4].

The results of the pre-analysis phase using VS soapUI show that 55.20% of the injected scripts are classified as successful attacks. Applying a set of rules to the results of the VS, the percentage drops to 16.49%, with a large number of false positive and false negative. In the second phase, we use the IF WSInject, obtaining 41.76% of successful attacks. However, 100% of Web Services presented vulnerabilities to this type of attack. In both phases, the use of WS-Security reduced significantly the number of vulnerabilities generated by Malformed XML.

Finally, this paper is organized as follows: Section 2 introduces the concepts and technology of Web Services, in addition to the security challenges. Section 3 presents the proposed approach and experimental study, while describing the rules to classify the attacks, and the obtained results. Section 4 concludes this paper, showing the main contributions and directions for future work.

II. SECURITY IN WEB SERVICES

The security violations in the systems occur due to the exploitation of existing vulnerabilities, which are faults introduced intentional or accidentally, during system

development. There are numerous causes of vulnerabilities, among which we mention the complexity of the systems, as well as the absence of a mechanism for verification of entries received. An attack exploits vulnerabilities, maliciously or not, and can compromise the security properties. The result of a successful attack is an intrusion into the system [6].

A. WS-Security and Their Specifications

Security in Web Services can be treated as point-to-point or end-to-end [7]. Different standards were proposed for each context. Point-to-point aims at ensuring the security during the transport of data. In this sense, there are several standards such as HTTPS, an extension of HTTP. The point-to-point security ensures the confidentiality of the data transported, but in the case where messages pass through intermediate terminals, before reaching the final destination, the security of the messages is not guaranteed.

End-to-end security aims at protecting the exchange of SOAP messages among clients and servers, encrypting the information from source to destination, even with the existence of intermediates. Among the proposed specifications, *XML Signature* can be mentioned [8], which defines rules for generating and validating digital signatures expressed in XML. *XML Encryption* [9] specifies the process of data encryption and its representation in XML, while *Security Token* [10] proves the identity of the client, so they can have access to services, using security credentials. There is also a OASIS specification, the *WS-Security* (*WSS*) [11], which defines a set of extensions to SOAP specifications and uses XML-Signature, XML-Encryption and Security Token to provide: (1) *integrity*, using digital signatures for all or part of the messages; (2) *confidentiality*, allowing SOAP messages to be encrypted partial or totally; (3) *authenticity* and *authorization* using security credentials in the SOAP messages.

Beyond security of messages, there is other gamma of applications based on Web Services, such as resource protection and security policies. Since our interest is in the WS-Security, we will not address these issues.

B. Security Testing

The Security Testing [6], allows to evaluate vulnerabilities in applications and services of different types of security attacks, e.g. Malformed XML, and discover new vulnerabilities before they are exploited by attackers. For this purpose there have been developed a variety of techniques, tools and languages classified into static and dynamic techniques.

Static techniques analyze and inspect the code. They are anticipated detection techniques, which carry many benefits such as reduced cost and time. There is no need to execute the services to apply this technique, differently from the dynamic ones. In the dynamic technique we have Penetration Testing, Fuzz Testing and Fault Injection.

Fault Injection consists of introducing, either by hardware or software, fault or errors in a system and observe its behavior [12]. There are several ways to inject faults. The most attractive is by software, in which faults are introduced by an injector, responsible for injecting faults in the system, before or during execution. In this technique, the tests are composed of two input sets, *workload* and *faultload*. The first represents the entries that activate the server, while the second represents the faults to be introduced.

There are numerous studies in the literature proposing the use of this technique to test application security: In [13] a fault injector is used to test security on firewalls and intrusion detection systems, simulating attacks on TCP/IP. [14] uses this technique to test a security protocol used in mobile communication devices on the Internet. For security testing of Web Services, there has also numerous works, among which we mention [15] and [16], which use perturbations in the SOAP messages to emulate attacks, similar to our proposal. However, these studies use specific injectors for one type of attack, while our injector is a general purpose one. Furthermore, these works only apply to message corruption, while in our case other types can be performed, for example, delayed delivery of messages.

III. MALFORMED XML AGAINST WS-SECURITY

In this section we first show the architecture of tests used and present, step-by-step, the execution of the experiments.

A. Test Architecture

The proposed approach uses fault injection as a technique for security testing. For this purpose we used an injector, developed in a previous work [17], called WSInject. This tool acts as a proxy between the client and server, allowing to inject communication faults for both service testing and composition testing. The aim was to test services in isolation. In this case, the injector intercepts the requests sent by the client through SOAP messages, before being forwarded to the server, as illustrated in Figure 1.

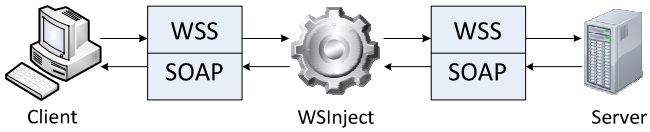

Figure 1. Test architecture used.

Since WSInject behaves as an HTTP proxy server, its configuration is easy to perform. Interception and modification of messages exchanged between the client and server is transparent. Thus, the tool does not need the source code of the service, or interfere with the execution platform, which makes it possible to be used by both providers and users of the service.

WSInject uses scripts to describe the faults to be injected, which are text files containing one or more *Fault Injection Statements* (commands). Each *Fault Injection Statements* consists of a *ConditionSet* and a *FaultList*. The *Fault Injection Statements* is a kind of *condition-action* command. By intercepting a message and satisfy a set of conditions, the fault list is injected. Figure 2 shows an example of executable script. In bold we have the keywords that specify *conditions* and *actions*. The first line contains a condition and two actions, in which case, each time the URI make a call to the Web Service

and its response contains the string "Hotel", all occurrences of "Name" are replaced by "Age" and a duplicated content is generated. In the second line, every time a message contains the string "caught exception" and is a response, the message content is empty.

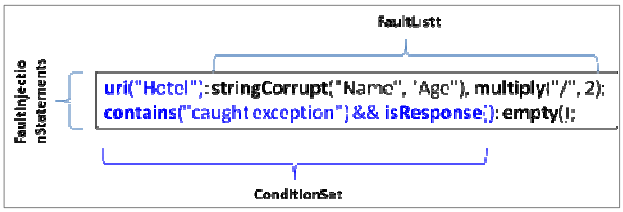

Figure 2. Samples script.

Once defined the architecture of tests, in order to perform the attacks, the following steps are needed:

1. Preparation: in this step we define the attacks, the *faultload* to be used, and select the Web Services that will be tested.
2. Execution: This step aims at setting the workload to generate SOAP message traffic, close to the reality. Furthermore, the generated data will be analyzed.
3. Analysis of results: analyzes data collected in the previous step and determines the existence of vulnerabilities.

B. *Preparation*

Although it was possible to emulate various types of attacks with the IF WSInject, this paper presents the tests using the Malformed XML attack, which causes the Web Service to expose its confidential information and generate fault in the target system (crash) [2]. This vulnerability occurs when a Web application does not validate the information received from external entities and processes the SOAP message, generating fault on the server. For example, in Figure 3, the client sends a form to be processed by the Web Service.

XML message request

```
1:   <body>
2:     <form>
3:       <first_name>Alice</first_name>
4:       <last_name>Kidman</last_name>
5:       <email>alice.kidman@email.com</email>
6:       <comment>write your comments here!</comment>
7:       <input type="submit">submit</input>
8:       ...
9:     </form>
10:  </body>
```

Figure 3. Example of XML form with user information.

A Web application that does not validate the information, allows the attacker to send the following comment:

```
<comment><xml>xml<joke></xml1234</comment></joke>
```

With this modification, the attacker inserts malformed XML fragments, leaving elements without declaration and open labels. The variations of the attack are extensive. Moreover, an effective attack gives the attacker enough information to perform other attacks, e.g. routes file, type of database and programming languages used, among others.

To emulate this attack, the injector must intercept SOAP messages, recognize the transactions described in WSDL, and corrupt the values of the parameters. To set the values to be used, we rely on information from the literature, as well as the attacks produced by the add-on Security Testing VS soapUI. Examples of the scripts generated are shown in Table I.

TABLE I. SAMPLE SCRIPTS USED TO EMULATE MALFORMED XML

WSInject scripts
isRequest():stringCorrupt("<ujc:filterdCalId>555</ujc:filterdCalId>","<ujc:filterdCalId><xml>xml<joke></xml></joke>555</ujc:filterdCalId>");
isRequest(): stringCorrupt("<ujc:filterdCalId>555</ujc:filterdCalId>", "<UJC:FILTERDcALiD>555&</UJC:FILTERDcALiD>");
isRequest(): stringCorrupt("<ujc:filterdCalId>555</ujc:filterdCalId>", "<ujc:filterdCalId newAttribute=\"XXX\">555</ujc:filterdCalId>");
isRequest(): stringCorrupt("<ujc:filterdCalId>555</ujc:filterdCalId>", "<ujc:filterdCalId xmlns:ujc=\"http://.../hello.jsp\">555:filterdCalId>");
isRequest(): stringCorrupt("555</ujc:filterdCalId>", "</ujc:filterdCalId>ujc:filterdCalId>555</ujc:filterdCalId>");

In Table I, the script uses the condition isRequest () which selects requests from a client to a server. In each request the injector uses stringCorrupt to replace the occurrences of the operation <ujc:filterdCalId> by the Malformed XML attack, e.g. <ujc:filterdCalId><xml>xml<joke></xml></joke>555 </ujc:filterdCalId>, with the objective of generating faults in Web Services.

The WS-Security provides protection against this type of attack, using Security Tokens (security credentials) with authentication of the client information (cf § II.A). The security information is embedded within tags <wsse:Security>, where each SOAP message can contain one or more tags. Given that a SOAP message can pass through several intermediate Web Services, WS-Security allows them to just read or modify parts of the message that they are directed to. In Figure 4, the Web service sender is informed that the client, properly authenticated, sends the request, as shown in lines 5 and 6.

XML message with WS-Security and Security Tokens

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

Figure 4. Example of SOAP message with Security Token (security credentials) to request a Web Service.

As part of the Preparation phase, were made experiments of pre-analysis, which allowed: i) select a set of Web Services to be tested; ii) determine the *faultload* to be injected and the communication scheme, which included the identification of entry points (operations and parameters) described in WSDL; and iii) identify the behavior of the Web Service in presence of faults that characterize a successful attack.

For this pre-analysis, we selected 10 Web Services from a set of 22,272, obtained from the UBR (Universal Business Registry) Seekda, only five of which use the WS-Security standard. These services have properties required to reproduce the attack, such as authentication operations and the use of WS-Security.

C. *Execution*

An important aspect of testing Web services is the generation of real network traffic - *workload*. It represents the requests that activate the Web Service. To be as realistic as possible during testing, it should generate traffic the closest as possible to the actual flow of SOAP messages. To generate the *workload* we used the add-on *Load Testing* of VS soapUI, representing the client, shown in Figure 1. The generated traffic consists of requests made to Web Services with purpose to emulate a real client.

The Figure 5 illustrates the injections campaign for tested Web Services. For each Web Service were developed 5 injection scripts, each one specifying the corruption of a determined parameter value and its operation, as shown in Table I. For each script, the workload sends 100 requests. In total, 5000 attacks were carried out.

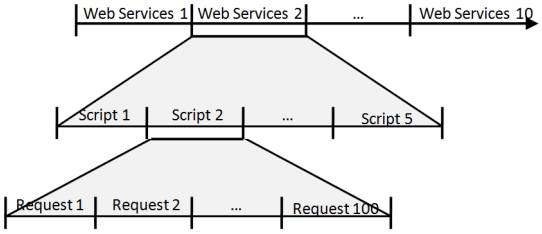

Figure 5. Campaign of injection attack of Malformed XML.

Given the large number of combinations of operation values and parameters for all Web Services, it is unviable to generate all the attacks needed to corrupt them. For this reason, we chose to make only a subset of these. The pre-analysis experiments have allowed us to determine the operations and parameters that would be interesting to use as target, i.e. we injected in those more likely to defect (fail), in order to decrease the amount of false positive present in VS tools.

D. *Analysis of Results*

An important aspect of this step is to identify when a vulnerability was effectively detected, i.e. when an attack was successful, excluding potential false positives. We must differentiate when an invalid result is obtained due to an internal failure (unintentional) to the Web Service or if it is consequence of a successful attack.

Given the approach proposed of black box to determine if an attack was successful, we used, as sources of information, the logs stored by the tools (VS soapUI and IF WSInject). The logs contain requests made by the client and the responses sent by the server. Figure 6 shows an example of the log produced by WSInject, in which the differences are indicated in the request (modified after the corruption of the parameter) and the response sent by the service. In this case the attack was successful, because the server returned code 500 Internal Server Error. In addition, we included the status codes from responses sent by the HTTP protocol, for example the error code 200, which indicates that the server did not detect the attack. These results were obtained by analyzing the log produced by WSInject described in Figure 6 (script 1).

isRequest():stringCorrupt("555</ujc:filterdCalId>","</ujc:filterdCalId>ujc:filterdCalId>555</ujc:filterdCalId>");

Request	Response
1: <soapenv:Envelope.....>	1: **HTTP/1.1 500 Internal Server Error**
2: <soapenv:Header/>	2: X-AspNet-Version: 2.0.50727
3: <soapenv:Body>	3: <?xml version="1.0" encoding="..."?>
4: <ujc:GetFilteredCalendarByID>	4: <soap:Envelope... xmlns:xsi="...">
5: <ujc:filterdCalId>	5: <soap:Body>
6: **</ujc:filterdCalId>**	6: <soap:Fault>
7: **ujc:filterdCalId>555**	7: <**faultcode**>soap:Client</**faultcode**>
8: **</ujc:filterdCalId>**	8: <**faultstring**>**Server was unable to read request**. ---> **There is an error in XML document (5, 47)** </**faultstring**>
9: </ujc:GetFilteredCalendarByID>	9: <detail />
10: </soapenv:Body>	10: </soap:Fault>
11: </soapenv:Envelope>	11: </soap:Body>
	12: </soap:Envelope>

Figure 6. Example of log generated by WSInject.

From this information we analyzed the responses of Web Services, applying a set of rules that allowed us to identify when vulnerability was effectively detected by the security testing and when an attack was properly rejected by the security specification WS-Security. This step is critical to reduce the number of false positives, i.e. avoid indicating vulnerability when it does not exist. We used the list of HTTP status codes described in Table II.

TABLE II. LIST OF HTTP STATUS CODES

HTTP codes	**Description**
200 – OK	Standard response for successful HTTP requests. Vulnerability confirmed because the system executed the request without detecting the attack.
400 – Bad Request	The request cannot be answered due to bad syntax. We considered unsuccessful attack because the server detected the attack.
500 – Internal Server Error	The server failed to comply with an apparently valid request or encountered an unexpected condition which prevented it from answer the request made by the client. Consider analyzing the response from the server using the tag <soap:Fault> within the body of the message. This tag provides error and status information of the SOAP message containing the sub-elements: • <faultcode>, the identification code of fault. • <faultstring>, readable explanation of the fault. • <faulttactor>, information on who/what caused the fault.

HTTP codes	Description
	• <details>, detailed information about the error. Faultcode values can be classified into four types: • *VersionMismatch*: The server has encountered an invalid namespace in the SOAP message envelope. • *MustUnderstand*: The fault of MustUnderstand indicates the absence of a mandatory element in the SOAP message header. • *Client*: The message was structured incorrectly or contains incorrect information. • *Server*: The server has a problem, so that the message cannot be processed.

In the pre-analysis phase, we generated the set of rules. First, the services were run without fault injection in order to evaluate the responses gotten. Then attacks were introduced using a VS soapUI with Security Testing add-on.

Based on the results obtained in the pre-analysis and interpretation of the HTTP status codes given in Table II, we created a set of rules to evaluate the results, similar to [18], obtaining a set of 13 rules, described in Table III.

TABLE III. RULES TO IDENTIFY SUCCESSFUL ATTACKS ON WEB SERVICES

#	Description
1	If the response message received contains "HTTP 200 OK" status and the system executed the request without detecting the attack, then there is a successful attack.
2	If the response message received contains "HTTP 200 OK" status and responded with a robust message, describing the existence of an error in the request, then there is a detected attack.
3	If the response message received contains "HTTP 400 – Bad Request" (e.g. Request format is invalid: Missing required soap: Body element.), then there is a detected attack.
4	In the absence or presence of attacks, if the response received contains message with code "HTTP 500 Internal Server Error", then there is software failure, because the response was not caused by the attack, but a software error.
5	If in the absence of attacks, the response received contains message code "HTTP 500 Internal Server Error" and in the presence of attack an "HTTP 200 OK", then there is a successful attack.
6	If in the absence of attacks, the response received contains message code "HTTP 500 Internal Server Error" and in the presence of attack the code "HTTP 400", then there is a detected attack.
7	If the response message contains code "HTTP 500 Internal Server Error" and there was no dissemination of information and the response describes the existence of error in the request, then there is a detected attack.
8	If the response message contains code "HTTP 500 Internal Server Error" and there was dissemination of information (e.g. software, programming language, library functions, database), then there is a successful attack.
9	If the response message contains code "HTTP 500 Internal Server Error" and route directory (Path) from the server or detailed information about the user's connection (session cookies, cryptosystems techniques used like SSL) were disclosed, then there is a successful attack.
10	If it is possible to redirect the user to other Web Services or access unauthorized pages, then there are successful attacks.
11	If it is possible to execute part of codes (e.g. Java script) or system calls on the server, then there are successful attacks.
12	If the server times out, it is considered server failure (crash) and there is a successful attack.
13	If none of the above rules can be applied, then the result is considered inconclusive, because there is no way to confirm the existence of vulnerability.

The add-on Security Testing for VS soapUI analyzes the presence of vulnerabilities on the type of attack Malformed XML. The results are described in Table IV. Note that 38.71% of the attacks were classified as false positives, compared to 16.49% of successful attacks. In the Figure 7, it was observed that the use of WS-Security with Security Tokens increases the detection of attacks in the services tested from 18.02% to 71.96% (false positive) and decreases the incidence of vulnerabilities from 22.09% to 7.48% (successful attacks).

TABLE IV. RESULTS OF THE PRE-ANALYSIS PHASE

Malformed XML Attacks with VS soapUI					
Web Services	***Total Attacks***	***False Positive***	***Detected Attacks***	***False Negative***	***Successful Attacks***
Without WSS	**172**	31	53	50	38
% attacks inj.	**100%**	18.02%	30.81%	29.07%	22.09%
With WSS	**107**	77	10	12	8
% attacks inj.	**100%**	71.96%	9.35%	11.21%	7.48%
Total Attacks	**279**	108	63	62	46
% attacks inj.	**100%**	38.71%	22.58%	22.22%	16.49%

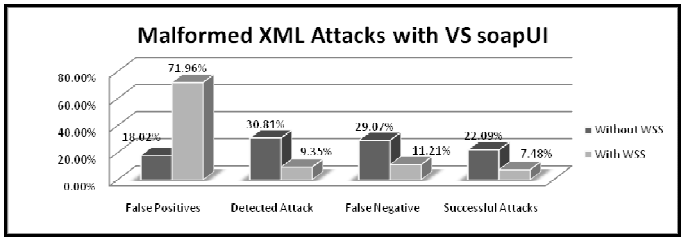

Figure 7. Campaign with VS soapUI.

In the phase of fault injection, we use the IF WSInject to emulate Malformed XML attacks. The results are summarized in Table V. It is important to note that this tool was successful to inject all the 5000 attacks. In Figure 8 we observed a meaningful improvement in the detection of attacks by the use of WS-Security, from 36.48% to 80.00%. This huge difference occurs because of security credentials (Security Tokens), which verify the authenticity of the client, among other parameters. The 20.00% of successful attacks with WS-Security correspond to two types of system vulnerabilities: (1) the service processes the request, which returns the HTTP code 200 for the execution of the message, and (2) the message provides information that can be used for other attacks, describing syntax problems (code 500).

TABLE V. RESULTS OF THE INJECTION PHASE

Malformed XML Attacks with IF WSInject			
Web Services	***Total Attacks***	***Detected Attack***	***Successful Attacks***
Without WSS	**2500**	912	1588
% attacks inj.	**100%**	36.48%	63.52%
With WSS	**2500**	2000	500
% attacks inj.	**100%**	80.00%	20.00%
Total Attacks	**5000**	2912	2088
% attacks inj.	**100%**	58.24%	41.76%

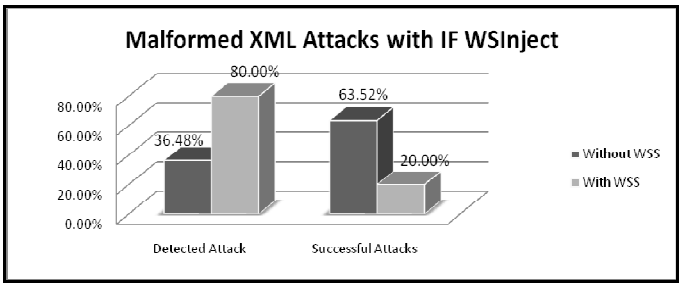

Figure 8. Campaign with IF WSInject.

As can be seen, the use of IF WSInject increases the number of successful attacks against the VS soapUI from 16.49% to 41.76%. With the use of WS-Security with Security Tokens, the number of successful attacks was reduced for both cases, with VS soapUI from 22.09% to 7.48%, and with IF WSInject from 63.52% to 20.00%.

CONCLUSION AND FUTURE WORK

The use of WS-Security improves the detection of Malformed XML, a type of injection attack, which causes fail on the target system and allows it to find vulnerabilities that could be exploited by other types of attacks. The results emphasize the fragility of the systems based on Web Service and the considerable importance of security mechanisms like those described in this paper.

The use of WSInject as a tool to inject Malformed XML is an advantage of the proposed approach, which can be used to emulate various types of attacks, being able to generate variants of them, which is usually limited in the tools commonly used to test security such as the vulnerability scanners.

This type of research is very important, both for the development of protection techniques and the development of security testing, because while the absence of fault by definition is undemonstrable and not robust, the presence of a system fault is demonstrable.

As future work, we intend to combine different types 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 Gabriela Batista Leão, from the Laboratory of Computer Networks (LRC), at the same institute, for her collaboration throughout the current work.

REFERENCES

[1] J. Meiko, G. Nils, H. Ralph and L. Norbert, “SOA and Web Services: New Technologies, New Standards - New Attacks,” Web Services, 2007. ECOWS '07. Fifth European Conference on , vol., no., pp.35-44, 26-28 Nov. 2007.

[2] A. Ghourabi, T. Abbes and A. Bouhoula, “Experimental analysis of attacks against web services and countermeasures,” In Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services (iiWAS '10). ACM, New York, NY, USA, 195-201, 2010.

[3] V. Patel, R. Mohandas and A.R. Pais, “Attacks on Web Services and mitigation schemes,” Security and Cryptography (SECRYPT), Proceedings of the 2010 International Conference on , vol., no., pp.1-6, 26-28 July 2010.

[4] Eviware. soapUI, the Web Services Testing tool – Security Testing Tool –. Obtained in Aug/2011 at: http: //www.soapui.org/.

[5] M. Salas, and E. Martins, “Emulação de Ataques do Tipo XPath Injection para Testes de Web Services usando Injeção de Falhas,” Workshop de Testes e Tolerância a Falhas (WTF), (30/04/2012 a 30/04/2012), Ouro Preto, MG, Brasil.

[6] C. Cachin, and J. Camenisch, “Malicious and accidental-fault tolerance in internet applications: Reference Model and Use Cases,” LAAS, MAFTIA, 2000.

[7] IBM “Security in a Web Services world A proposed architecture and roadmap,” Whitepaper, April 7, 2002, V1.0.

[8] D. Eastlake, et al , “XML signature syntax and processing,” 2nd Edition, 2008.

[9] D. Eastlake, J. Reagle, T. Imamura, B. Dillaway, E. Simon, “XML encryption syntax and processing,” W3C Recommendation, 2002.

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

[11] K. Lawrence, C. Kaler, A. Nadalin, R. Monzillo, P. Hallam-Baker, “Web Services Security: SOAP message security 1.1 (WS-Security 2004),” OASIS.

[12] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J-C. Fabre, J-C. Laprie, “Fault injection for dependability validation: a methodology and some applications,” 1990.

[13] P.C.H. Wanner, and R.F. Weber, “Fault injection tool for network security evaluation,” Dependable Computing, 2003.

[14] A. Morais and E. Martins, “Injeção de ataques baseados em modelo para teste de protocolos de segurança”. Instituto de Computação, UNICAMP, 2009.

[15] M. Vieira, N. Antunes, and H. Madeira, “Using Web Security Scanners to Detect Vulnerabilities in Web Services,” Conf. on Dependable Systems and Networks, 2009.

[16] C. V. Ana de Melo, P. Silveira, “Improving data perturbation testing techniques for Web Services,” Inf. Sci. 181, 3, 600-619, 2011.

[17] A. W. Valenti, W. Y. Maja, and E. Martins, “WSInject: a fault injection tool for Web Services”. Instituto de Computação, UNICAMP, 2010.

[18] N. Antunes, and M. Vieira, “Detecting SQL Injection Vulnerabilities in Web Services,” Dependable Computing, 2009.LADC '09. Fourth Latin-American Symposium 2009.