The Journal of Systems and Software 109 (2015) 18–31

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Assessing the security of web service frameworks against Denial of Service attacks

Rui André Oliveira, Nuno Laranjeiro*, Marco Vieira

CISUC, Department of Informatics Engineering, University of Coimbra, Coimbra, Portugal

ARTICLE INFO

Article history:
Received 29 July 2014
Revised 28 June 2015
Accepted 2 July 2015
Available online 14 July 2015

Keywords:
Security
Web service frameworks
Experimental assessment

ABSTRACT

Web services frequently provide business–critical functionality over the Internet, being widely exposed and thus representing an attractive target for security attacks. In particular, Denial of Service (DoS) attacks may inflict severe damage to web service providers, including financial and reputation losses. This way, it is vital that the software supporting services deployment (i.e., the web service framework) is able to provide a secure environment, so that the services can be delivered even when facing attacks. In this paper, we present an experimental approach that allows understanding how well a given web service framework is prepared to handle DoS attacks. The approach is based on a set of phases that include the execution of a large number of well-known DoS attacks against a target framework and the classification of the observed behavior. Results show that four out of the six frameworks tested are vulnerable to at least one type of DoS attack, and indicate that even very popular platforms require urgent security improvements.

1. Introduction

Web services (WS) are being deployed in diverse scenarios, ranging from small local businesses to more global scale business-to-business environments. SOAP-based web services consist of self-describing components that can be used by other software across the web in a platform-independent manner, are supported by standard protocols such as SOAP and WSDL (Curbera et al., 2002), and nowadays play a key role in the integration of heterogeneous systems. In a web services environment, the provider offers a well-defined interface to consumers, which includes a set of operations and typed input/output parameters.

A set of steps must be performed for a provider to offer a SOAP-based web service. First, a developer must code the **web service application**. After the service application is coded, it is deployed in the context of an **application server** (e.g., Apache Tomcat, JBoss AS). This requires the use of a **web service framework** (or stack) that, together with the application server, acts as a container for the web service. In the context of this combined set of software components, the application server delivers the SOAP messages arriving from clients (typically via HTTP) to the web services framework, which then processes (parses messages, checks for errors, and builds programming language-level objects) and delivers them to the web service application (U and Rao, 2005).

The web service framework is a key software component and handles all SOAP client requests including valid requests (those that are compliant with the WS specification), invalid requests (not compliant with the WS specification), and malicious requests (that try to exploit vulnerabilities in the web services frameworks and/or applications). Deploying a web service is currently a fairly simple process, as the supporting technologies are in a matured state and existing development tools provide easy ways to create services based on local (i.e., non-remote) software applications.

A key problem is that developers tend to assume that the maturity of the underlying web service frameworks, resulting from years of field experience, refined development, tests, and practical use, guarantees that the service applications are deployed in a stable and secure environment, which is not always the case. In fact, previous research shows that, although web services technology has reached a mature state and is designed based on a large amount of networking experience, WS frameworks are not more secure than any other network-based system (Jensen et al., 2009; Suriadi et al., 2010).

One of the main threats to the availability of the services deployed on the web are the well-known Denial of Service (DoS) attacks, which consist on overloading the server resources in a way that leads to legitimate users being denied to use a service (Ranjan et al., 2009). The authors in (Imperva, 2012) show that the target of this type of attacks is moving from the lower layers of the communication system (e.g., TCP/IP protocols) to the upper layers (e.g., the web

* Corresponding author. Tel.: +351 239 790 000.
E-mail addresses: racoliv@dei.uc.pt (R.A. Oliveira), cnl@dei.uc.pt (N. Laranjeiro), mvieira@dei.uc.pt (M. Vieira).

http://dx.doi.org/10.1016/j.jss.2015.07.006
0164-1212/

service). In fact, DoS attacks are more effective at the application level and, since many anti-DoS solutions target lower communication layers (Cloudfare, 2014; "Defeating DDOS Attacks," 2014), they often pass undetected.

Studies that try to understand how web service frameworks behave when facing DoS attacks are quite limited. Although some work has been conducted to detect vulnerabilities in web service applications and to understand how they can be exploited (Antunes et al., 2009; Duchi et al., 2014; Vieira et al., 2009) studies focusing on assessing the behavior of frameworks in the presence of DoS attacks are reduced to exploratory and isolated examples (Jensen et al., 2009; Suriadi et al., 2010). Furthermore, the existing tools that allow emulating WS attacks (e.g., penetration testers and fuzzers) are also limited, mostly targeting the service implementation itself and disregarding the potential vulnerabilities of the underlying framework ("soapUI - Functional Testing," 2012; "WSBang," 2012; "WS-Fuzzer Project," 2012). In fact, the tools that can be used to attack frameworks frequently include limited sets of attacks, but most of all, even though they can be used to attack some platform, they are not prepared to support the evaluation of WS frameworks, as they lack the description of the set of steps and metrics required to assess a given framework. Thus, tools become rather useless for both practitioners (e.g., to assess the security of their platforms) and researchers (e.g., to study frameworks in terms of different security properties).

In this paper we propose an experimental approach to evaluate the security of web service frameworks. The approach builds on the work presented in Oliveira et al. (2012a) and is based on a set of DoS attacks that have been compiled from security research studies, existing security tools, and field experience, and that target core message exchange features (i.e., communication involving common data types). Attacks focusing web services extensions (which provide features such as message integrity or confidentiality) are out of the scope of this work. In practice, we use the compiled attacks in combination with regular requests in a set of runtime tests (during three distinct phases) to assess the behavior of frameworks in the presence of such malicious requests. Observed failures are classified using an adaptation of the CRASH scale (Koopman et al., 1997) and dubious behaviors (that indicate abnormal allocation of system resources) are also analyzed. Comparing to the work in Oliveira et al. (2012a), we perform the following extensions: 1) we augment the approach with a technique for analyzing the impact of the attacks on frameworks in a quantitative manner; 2) we include the comparison of pairs of test periods (e.g., the comparison of the behavior observed in the presence of attacks with the one previously observed when executing non-malicious requests), which allows us to understand the impact of the attacks in a more detailed manner; and 3) we evaluate the latest version of a larger number of web service frameworks and for some cases we compare the results with the ones from previous versions (in order to understand the security evolution of the frameworks over time).

The experimental evaluation includes well-known and widely used frameworks, namely: Apache Axis 1, Apache Axis 2, Apache CXF, Oracle Metro, Spring JAX-WS, Spring-WS and XINS ("Apache Axis2/Java," 2012; "Apache Axis," 2006; "Apache CXF," 2012; "Metro," 2012; "Spring support for JAX-WS RI—Project Kenai," n.d.; "Spring Web Services - Home," 2013; "XINS - Open Source Web Services Framework," 2013). Results show that most of frameworks are quite resistant to the large set of security attacks executed by our tool, but the issues detected (e.g., high CPU/memory usage and unexpected exceptions) also indicate the potential presence of major security vulnerabilities in the frameworks, requiring urgent attention and corrective measures from developers. The quantitative analysis performed, highlighted previously unnoticed impact of attacks and disclosed additional security issues, even for frameworks that were tested in previous work (Oliveira et al., 2012a).

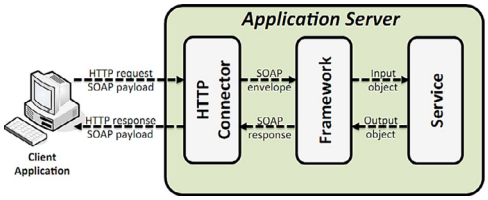

Fig. 1. Web services interactions and supporting infrastructure.

The paper is organized as follows. The following section presents background and related work. Section 3 describes the approach used to test the security of web service frameworks. Section 4 presents the experimental scenarios designed to demonstrate the proposed approach and Section 5 presents the results obtained for each of the tested frameworks. Section 6 discusses the quantitative impact of the attacks and compares the behavior of different versions of the same framework. Finally, Section 7 concludes the paper.

2. Background and related work

This section presents core concepts regarding the web services technology and supporting environment, and previous work on DoS attacks and framework-level vulnerabilities. The section concludes with an overview of the state of the practice, with focus on the limitations of current WS security testing tools.

2.1. Concepts on web services

In a web services environment a provider supplies a set of services to consumers (Curbera et al., 2002). The Simple Object Access Protocol (SOAP) is used for exchanging XML-based messages between the consumer and the provider over the network (usually using HTTP). In a typical interaction the consumer (the client) sends a request SOAP message to the provider (the server). After processing the request, the server sends back a response with the results. A service may include several operations and is described using WSDL (Web services Description Language) (Curbera et al., 2002). Restful Web services (Oracle, 2013), based on plain XML or JSON, are nowadays becoming quite popular also and represent another architecture being used in service-based environments. Despite sharing common characteristics (e.g., XML-based responses), this paper is specifically focused on SOAP-based services, and the adaptation of the proposed approach to other WS technologies is considered as future work.

As shown in Fig. 1, the core software components supporting SOAP web services and allowing end-to-end communication between clients and servers are: 1) a web services framework (e.g., Apache Axis or Metro); and 2) an application server (e.g., Apache Tomcat). In practice, a client sends a SOAP message via HTTP, the HTTP connector handles and processes the incoming HTTP request, retrieves the SOAP message and delivers it to the web service framework. The framework then processes and delivers the SOAP message to the actual service implementation (i.e., the web service application). In short, the framework validates each message and transforms it into an object that can be handled by the application. After this object is processed by the application, the reverse path is taken, with the return object being serialized into a SOAP response that is sent via HTTP to the client (U and Rao, 2005).

There are currently tens of web service frameworks available. Some can be used with their own proprietary application server (e.g., Axis), while others are included in enterprise application servers (e.g., Glassfish or JBoss AS) or can generally be added to more lightweight servers, such as Apache Tomcat or Jetty. In fact, these are very popular among developers and providers, as they deliver the typical core features that are required by web applications (e.g., HTTP request handling, servlet container, JSP engine, etc.) and are not loaded with

extraneous functionalities, which many times are not used. Tomcat is, indeed, a very popular solution (Zeichick, 2008), and its core engine is used by many enterprise servers, such as JBoss or Apache Geronimo.

2.2. Security testing tools

Among other solutions, Acunetix Web Vulnerability Scanner, HP WebInspect, IBM Rational AppScan are well-known security testing tools for web applications (Antunes et al., 2009). Despite their popularity, they focus on typical application-level vulnerabilities, such as SQL Injection or Cross-Site Scripting, thus being of little use for evaluating service frameworks. SoapUI ("soapUI - Functional Testing," 2012) is a testing tool that recently started supporting the execution of security tests in web services (details on the attacks supported by current testing tools are presented in Section 3). However, the source code is poorly documented, limiting its extensibility. WSFuzzer and WSBang are Python security testing tools for SOAP services ("WSFuzzer Project," 2012; "WSBang," 2012). WSFuzzer includes support for attacking web services using random attack strings with multiple parameters, and also allows testing attacks in web service frameworks (see Section 3 for details on specific attacks supported by this tool), however in WSBang tests are limited to random parameters.

WS-Attacker is an application that can be used to test web service frameworks (Mainka et al., 2012; "WS-Attacker," 2012). The concepts behind the tool and WS-Attacker's architecture are introduced in Mainka et al. (2012), including details about the technologies used and how to develop new plugins. Its extensibility is a strong point when the goal is to develop more complex security testing applications. WSFAggressor, the tool used in the present work (Oliveira et al., 2012b), is an extension of WS-Attacker and uses a partial set of its functions (e.g., the graphical interface). Comparing to WS-Attacker, we redefined the plugins system, implemented a larger set of attacks, and added support for the approach presented in Section 3. More details regarding WSFAggressor are available in Oliveira et al. (2012b).

2.3. DoS in web services

In general, a Denial of Service attack is an attempt to make a service unavailable to legitimate users (McDowell, 2013). In the specific case of WS frameworks, hackers frequently try to exploit the overhead that results from processing malicious SOAP messages (Gang et al., 2006) by building large and valid/invalid SOAP requests (Jensen et al., 2009). One key aspect regarding these XML-based DoS attacks is that most of them require minimum knowledge from the attacker (Jensen et al., 2009).

A study on the vulnerabilities and attacks on web service frameworks is presented in Jensen et al. (2009). Among others, two important XML based attacks are referred: Coercive Parsing and Oversized Payload. In Intel SOA Expressway (2006) an extensive threat model for XML content is presented, detailing attacks that extend the former two and including other types of attacks (e.g., based in large arrays). There are, in fact, several studies defining and classifying attacks, but few target their impact in practice.

A study is presented in Govindaraju et al. (2004) with the goal of characterizing the performance of SOAP frameworks. The work uses distinct arrays (including different sizes) to study the cost of the serialization and deserialization processes of XML parsers. The different parser implementation strategies are analyzed and the authors indicate that naïve implementations can lead to considerable processing time (which can be critical in DoS attack scenarios).

The causes for low performance observed in many XML-based applications (e.g., XML parsers and web services) are discussed in Gang et al. (2006). The authors conclude that parsing XML documents frequently generate intensive memory allocation operations and that the allocated objects are typically long-lived. Arrays occupy large portions of space during regular XML processing, thus being a problem when facing security attacks that target memory depletion.

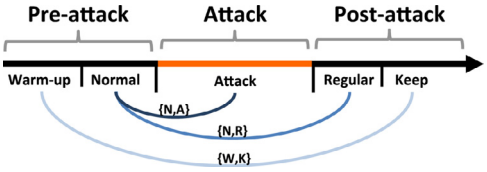

Fig. 2. Approach overview.

The impact of Denial of Service attacks on web service frameworks is studied in Suriadi et al. (2010). The experiments conducted include tests with Metro, Axis, .NET WCF, and Ruby, and are based on flooding attacks (requests sent in sequence and in parallel). Results indicate that attacks tend to impact CPU resources rather than memory. A key issue is that there is no reference in the paper to the specific versions of the platforms tested and containers used, which is a huge limitation on the reproducibility of the experiments, preventing us from comparing their results with our work.

An approach for automatic evaluation of the impact of Denial of Service attacks on web service frameworks (Falkenberg et al., 2013). The authors assume that there is no physical access to the machine being tested and, as such, the attack executor is limited to sending payloads and measuring response times. The approach is implemented as a plugin for the WS-Attacker tool and consists of two phases. In the first phase the tool sends regular (i.e., non-malicious) requests to the server and in the second phase the tool sends malicious requests to the server. In parallel, the tool simulates the presence of an additional client that sends regular requests to the server and collects the response time. A key limitation is that, besides only considering the response time metric, the approach disregards the impact of the attacks after the second phase.

In summary, previous studies indicate that web service frameworks are vulnerable to attacks. However, the limitations found in those studies combined with the fast-paced development of web services and the absence of adequate security tools for testing service frameworks, justify further research on this topic. Our approach tries to overcome the observed limitations and may be used by developers to disclose potentially severe issues, enabling also the comparison of different platforms.

3. Approach for assessing frameworks in presence of Denial of Service attacks

This section presents our multi-phase approach to evaluate the behavior of web service frameworks when facing DoS attacks. As an approach based only on attacking a web service with malicious requests does not provide a view of the service behavior that is accurate enough (with exception of very clear cases, where for instance the service platform crashes), we created a compound procedure that includes a set of distinct phases that allow understanding the regular behavior of the service (i.e., in the presence of normal requests), the behavior of the service in the presence of attacks, and the effects of an attack on the regular service behavior (by comparing the behavior observed after the attack with the observations before the attack).

Fig. 2 presents an overview of the approach, showing the sequence of phases (which we designate as Pre-attack, Attack, and Post-attack), and periods (Warm-up, Normal, Attack, Regular, and Keep) within each phase. Fig. 2 also shows important relations between pairs of periods (for instance {N, A} to represent the relation between the Normal and Attack periods). A detailed explanation is presented in the next paragraphs.

Table 1
Attack types and tools.

Attack	Description	Example	Tools
Coercive Parsing (CP)	SOAP body is set with a large quantity of nested open XML tags named after the operation arguments names	100,000 nested open XML tags named with the target operation argument name	WSFuzzer, WS-Attacker
Malformed XML (MX)	A combination of XML malformations in each malicious request (e.g., tags not closed, invalid characters)	Two interlaced tags; one tag open but not closed; one attribute open but not closed; invalid characters	SoapUI
Malicious Attachment (MA)	A large quantity of binary data is sent with the request	A 100MB gziped binary file (randomly generated)	SoapUI
Oversized XML (OX)	The malicious request includes three types of oversized XML components: 1) Large XML tag names; 2) Large values enclosed in regular tags; 3) Large attribute names	Alternate invocations of the following request configurations: 1) XML tag oversized until a total size of 1.9Mb is reached; 2) XML tag filled with one regular attribute repeated until a total size of 1.9Mb is reached; 3) XML tag with a large attribute name (until 1.9Mb)	WS-Attacker
Soap Array attack (SA)	A large number of regular XML elements (e.g., a million elements)	1000,000 regular XML elements with a 6-byte String as value	WS-Attacker
Repetitive Entity Expansion (REE)	Compact recursive definition of DTD entities, which the XML parser expands into a set of large entities.	The expansion of a 100 references to a 3-byte entity, with each reference defined in terms of the previous one, expanding to $(2^{101} - 1) * 3$ bytes, i.e., requiring 7e + 21Gb in memory	-
XML Bomb (XB)	Combination of three types of requests that include: 1) Definition of a large external DTD entity (e.g., 100Mb) that is loaded by the framework; 2) A large entity (hundreds of Kb) is defined and referenced thousands times in sequence in the request; 3) A compact recursive definition of DTD entities, which the XML parser expands into a large set of entities	A combination of malicious requests sent alternately that include: 1) A 92.2Mb external entity; 2) 30,000 references to a 100Kb entity; 3) A billion references to 3-byte entities defined recursively in less than 1Kb but expanding to nearly 3Gb in memory	SoapUI
XML Document Size (XDS)	A valid large SOAP header or body (Mb size)	Requests including (sent alternately to the server): 1) A valid 1.9Mb SOAP header; 2) A valid 1.9Mb SOAP body	–
XML External Entities (XEE)	Requests that reference well-known system files	A request referencing: /etc/password; /etc/shadow; C:\boot.ini; C:\windows\System32\MRT.exe; and /dev/random	WS-Attacker, WS Fuzzer

1) **Pre-attack phase:** includes two periods, a **Warm-up** period, where no requests are sent to the service, and a **Normal** invocation period, where valid requests (non-malicious) are sent to the web service. The goal of this phase is to understand the behavior of the service framework when idle and when handling normal requests.
2) **Attack phase:** is composed of a single period (Attack), where malicious requests (of a given attack type) are sent to the web service provider. This phase is carried out with the purpose of understanding how the service framework behaves when facing security attacks.
3) **Post-attack phase:** includes two periods, a **Regular** invocation period, where non-malicious valid requests are sent to the server, and a **Keep** period, in which no requests are sent to the provider. The goal of this phase is to study if the attack phase has any effect on the regular service framework operation.

During these three phases, which have configurable durations, several parameters that represent the state of operation of the server are monitored, such as used memory, CPU usage, and number of allocated threads (Gang et al., 2006; Suriadi et al., 2010). In this work we focus on studying service frameworks in the presence of DoS attacks delivered as synchronous non-parallel requests. The (more complex) study of the behavior of frameworks in the presence of asynchronous, parallel or distributed requests is out of the scope of this paper. The following sections present each phase in detail.

3.1. Pre-attack phase

This phase is composed by two distinct periods: warm-up and normal. The **warm-up** period leads the service platform (i.e., the application server) to start up and allows collecting data regarding the idle behavior of the server (e.g., amount of used memory, number of allocated threads, CPU usage). Comparing this information with the Keep period of the Post-attack phase (relation {W, K}), helps understanding if a particular attack type affects the server behavior, even when there are no requests to handle. The **normal** invocation period serves the purpose of providing data regarding the simple operation of the service platform (i.e., when handling non-parallel requests). During this phase, valid requests are sent to the server and runtime data are collected. These data allow later checking if a given attack impacts the service or not (relation {N, A}), or if its effects are visible even after the server has handled the attack (relation {N, R}).

3.2. Attack phase

In this phase, a set of malicious requests implementing a given attack type is delivered to the service framework. The goal is to understand, not only what is the direct impact of attacks on a framework, but also to observe the behavior of the framework in the presence of attacked and then compare that behavior with the one previously observed when handling normal requests (i.e., non-malicious). This is represented in Fig. 2 with relation {N, A} that links the Normal and the Attack periods. We carried out a study to identify and select attacks designed to target WS frameworks, focusing not only on the research carried out in the web services security area (Intel SOA Expressway, 2006; Jensen et al., 2009; Suriadi et al., 2010), but also on software applications that can be used for testing and attacking web services (Falkenberg et al., 2013; "soapUI - Functional Testing," 2012; "WSFuzzer Project," 2012).

Table 1 describes the attack types collected and lists the current security tools that implement each attack type. All attacks are implemented and available for use in our tool—WSFAggressor (Oliveira et al., 2012b).

As an illustrative example, we present the SOAP Array attack (one of the attacks used in this work) in Fig. 3. As shown, the goal of this attack is to create and send an array with a large number of elements.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:myw="http://MyWebService.dei.uc.pt">
   <soapenv:Header/>
   <soapenv:Body>
      <myw:getArray>
         <myw:param>attack</myw:param>
         <myw:param>attack</myw:param>
         <myw:param>attack</myw:param>
         <myw:param>attack</myw:param>

         <!-- repeat the previous line N times -->

      </myw:getArray>
   </soapenv:Body>
</soapenv:Envelope>
```

Fig. 3. The Soap Array attack.

A vulnerable framework will reserve memory space to handle that large array, likely leading to memory exhaustion of the attacked system (Jensen et al., 2009).

3.3. Post-attack phase

During this phase we observe the behavior of the service framework after being attacked. In some cases, the previous Attack phase is enough to cause a clear failure of the server; for instance, turning it unresponsive. This means that the Post-attack phase cannot be performed, since the server is unavailable to process requests. However, in other cases it is possible that an attack phase affects the server in a different way by, for instance, decreasing the amount of available memory. In these situations, the use of a Post-attack phase is crucial.

The Post-attack phase consists of a **Regular** invocation period and a **Keep** period. The former is useful to observe if the attack phase indeed had any effects on the service platform, namely if it is still capable of handling valid requests in a normal way (this is represented by relation {N, R} in Fig. 2). In order to verify this, we analyze the data collected during the Regular period and compare them with the data collected previously in the Normal period of the Pre-attack phase. Ideally, there should be no observable difference between these periods. The Keep period is used to detect if the service platform shows any dubious behavior when not handling requests anymore. It is also useful to compare the behavior of this period with the one observed for the Warm-up period (this is represented by relation {W, K} in Fig. 2), which allows understanding if, even after the attack phase finished and the execution of normal requests stopped, the framework shows signs of being affected by the attacks (e.g., by not being able to release previously allocated memory).

3.4. Classifying failures

As software systems may fail in distinct ways, it is useful to classify and group failures, thus enabling the comparison of distinct systems (Koopman et al., 1997). In this work we adopt the CRASH severity scale, which has been originally applied with success in the operating systems domain (Koopman et al., 1997), and also more recently in the web services domain (Vieira et al., 2007). The scale is summarized in the following points:

- **Catastrophic:** the service becomes corrupted, or the server or operating system crashes or reboots;
- **Restart:** the service becomes unresponsive and must be terminated by force;
- **Abort:** an abnormal termination is detected when executing the service (e.g., an unexpected exception);
- **Silent:** no error is indicated by the service on an operation that cannot be concluded or is concluded in an abnormal way;
- **Hindering:** the returned error code is incorrect.

Table 2
Infrastructure supporting the experiments.

Node	Software	Hardware
Client	Ubuntu 12.04 (32-bit) OpenJDK 1.6.0_20	Intel core 2 duo T6500 (2.1 GHz) 3Gb RAM
Server	Windows XP (64-bit) Oracle Java 1.6.0_30-b12	AMD Athlon X2 Dual Core 4200 (2.21 GHz). 4Gb RAM

As mentioned before, in some cases we may not be able to clearly identify failures. For example, when testing a framework, we may observe some fluctuation of the system parameters (high memory or CPU usage), yet the service provider continues to operate (i.e., it does not fail). In our approach, such **dubious behaviors** are subjected to further analysis, where the goal is to understand, in a quantitative manner, how different is the observed behavior from the normal one (e.g., the memory usage might duplicate while processing regular requests as a consequence of a previous attack, but still the service provider continues to operate).

4. Experimental scenario

The experimental setup consisted in deploying a client running WSFAggressor (Oliveira et al., 2012b) to attack a server configured with a selected set of service frameworks. The two test nodes (client and server) were setup into separate machines connected using an isolated Fast Ethernet network. Table 2 describes the nodes in terms of hardware and supporting software infrastructure.

As we can see in Table 2, we did not assess frameworks in combination with multiple operating systems. Although such setup might help providing a broader view of the behavior of the frameworks (including its combination with different operating systems), the focus of this paper is on the design of the proposed approach and on showing its usefulness. Providing more results is simply a matter of extending the experiments (which does not change the overall approach). We also do not consider a different network topology (i.e., more machines) as the focus is on the basic behavior, using direct, isolated interactions with services. The frameworks and remaining configurations are discussed in the following sections.

4.1. Web service frameworks

To demonstrate the proposed approach we selected seven well-known web service frameworks). As container (i.e., application server) we selected Tomcat 7.0.23 ("Apache Tomcat," 2012) due to its large use and popularity among developers and providers (Zeichick, 2008). The frameworks considered are **Metro, Apache CXF, Apache Axis 2, Apache Axis 1, Spring JAX-WS, Spring-WS,** and **XINS** ("Apache Axis2/Java," 2012, p. 2; "Apache Axis," 2006; "Apache CXF," 2012; "Metro," 2012; "Spring support for JAX-WS RI—Project Kenai," n.d.; "Spring Web Services—Home," 2013; "XINS—Open Source Web Services Framework," 2013). In addition to the latest stable versions

Table 3
Frameworks and XML parsers tested.

Framework name	XML parser	Latest version tested	Older version tested
Apache Axis 1	Xerces	1.4.1	–
Apache Axis 2	AXIOM	1.6.2	1.6.1
Apache CXF	Woodstox	3.0.3	2.5.1
Oracle Metro	Woodstox	2.3.1	2.1.1
Spring JAX-WS	Woodstox	1.9	–
Spring WS	Xerces	2.2.0	–
XINS	Xerces	3.1	–

we also tested older version of three of the frameworks (see last column of the table) in order to study whether issues found in older versions are fixed in the newest ones. Table 3 summarizes the selected frameworks, including the XML Parsers they use by default, as these play the key role of processing the messages passing between the container and the application.

Apache Axis 1 is a quite old and highly matured web service framework still used in many production systems. It is distributed with its own standalone server, although it can be deployed in other containers. Currently, there are no plans to introduce additional features to the latest version. Apache Axis 2 was designed with the goal of creating a more XML-oriented and modular platform that easily supports the addition of plugins to extend its functionality. Axis 2 can also be used with most popular servers or with its own standalone server.

Apache CXF is an open-source services framework that can be used to create SOAP web services, but can also use other protocols such as CORBA or RESTful HTTP. It can be deployed in a large variety of containers such as Tomcat, Jetty, JBoss AS, among others. CXF supports all of the latest usual web service standards, most notably the JAX-WS API (Sun Microsystems Inc., 2010).

Metro is an open-source web services stack whose development is managed by the Glassfish community, which is linked to Oracle Corporation. Metro is currently being bundled with the Glassfish server but, like most frameworks, can be used in other containers. The server choice depends on the specific requirements of each service and, as mentioned before, Tomcat is frequently the option due to the presence of core web application features and the absence of enterprise features that are only used in very specific cases.

Spring JAX-WS is a sub-project of the Glassfish project aiming at facilitating the deployment of web services using the Spring Framework. The main feature is that it supports the deployment of the service application on top of the Spring framework and thus allows the application to benefit from the advanced features of the Spring framework (e.g., injection of Spring beans, using handlers, custom transports).

Spring-WS is a project of the Spring community focused on creating "*contract-first SOAP service development*". This framework allows an easy integration with the highly popular Spring Framework and the focus is on allowing developers to focus on the service contract, by easily providing the means for creating services starting from the service description (i.e., the WSDL document). Spring WS provides easy support for using different XML parsers (e.g. Xerces, Axiom, JDom, Woodstox, etc…) and, if not explicitly configured, by default it uses the XML parser included in the Java Virtual Machine (Xerces).

It is worth noting that despite we are testing two Spring based implementations, for the context of our study they present substantially different characteristics. Spring JAX-WS is implemented based on the JAX-WS Reference Implementation allowing the automatic generation of the WSDL file that describes the service. This approach to web service development/deployment is commonly referred as "*Contract-last development*". On the other hand, Spring-WS requires the web service developer to design the XML schemas and the WSDL file that describes the service (besides the code implementation). Its approach to web service development is commonly referred as "*Contract-first development*".

Table 4
Test service design.

Operation name	Input	Operation Behavior	Output
getInt	Integer	Sets the value of the input parameter on the output parameter, without further transformations	Integer
getString	String	Calculates a hashcode over the input (using Java's *hashcode()*) and sets the output parameter with the resulting value	Integer
getArray	String Array	Sets the output parameter with the length of the incoming array	Integer
getFile	Data Handler	Calculates a MD5 hash over the incoming data	String

XINS is an open source Java-based WS framework that provides support for multiple protocols, including REST, XML-RPC, JSON, JSON-RPC and SOAP. XINS allows the user to specify a schema configuration that describes the service, but it also supports the automatic generation of the WSDL file.

4.2. Service design and configuration

In order to use and attack a WS framework we need to deploy a service application that allows exercising the system. Several choices have to be made when designing this service, in particular: the interface that each operation provides (type and number of input parameters and type of output parameter), and the work that should be performed by each operation (i.e., what should be the code executed in each invocation). We designed a test service that includes four operations with distinct input types. Table 4 summarizes the choices made during the design of the service (see the following paragraphs for a detailed description).

The design of this service's interface was inspired by performance benchmarks for web services, namely: Sun Microsystems WSTest 1.0, Microsoft's WSTest 1.5, and The Transaction Processing Council TPC-App ("Comparing Java 2 EE and .NET Framework," 2004; Microsoft, 2008; Transaction Processing Performance Council, 2008). Most WSTest operations use Integers, Strings, and Arrays as parameters. The names of the operations were defined based on the concatenation of the word "get" with the datatype supplied as parameter. The interfaces in TPC-App are also mostly based in these data types. Thus, we defined three service operations that use these common data types and added an operation that can receive a message attachment. This represents the cases where, for instance, a client sends a file to a server (e.g., an image or video).

As our target is not the web service application there are no representativeness requirements regarding the actual code for the services described above (we simply need an entry point to the framework, and this is provided by the web service interface). In fact, any specific function added to the service code would cause overhead (both in terms of CPU or memory usage), and we are interested in reducing such overhead so that the observed behavior is related, as much as possible, with the supporting platform, rather than with the service implementation. Note also that the decisions taken for designing the operations (described in Table 3) represent one possible setup and many other options are possible, such as distinct operations implementations, the use of extra data types, among others. Our goal was primarily to define a basic service entry point ("Comparing Java 2 EE and .NET Framework," 2004; Microsoft, 2008; Transaction Processing Performance Council, 2008) and then to implement useful operations (from the tests point-of-view) with minimal overhead.

Table 5
Client configuration.

Operation name	Input value
getInt	10
getString	6-byte String with random content
getArray	Two-hundred 6-byte Strings with random content
getFile	A 700Kb JPEG image file

4.3. Client configuration

In addition to the service, there is a client that is embedded in the WSFAggressor tool and is in charge of invoking and attacking each service operation. The choices associated with the client configuration can hence be divided in the options regarding the valid invocations and the ones related with malicious invocations (i.e., the attack configuration). These configuration aspects are discussed in the following paragraphs.

Before running the client we need to select the values used in each regular service call (i.e., calls that do not try to attack the service). In general, we selected the maximum values found in all operations defined by the WSTest and TPC-App benchmarks, so that the stack is exercised as much as possible, even with a regular request. Although of little relevance when considering, for instance, numbers (due to the small amount of bytes required to represent them), this can have particular relevance when a stack needs to deserialize an array, which essentially multiplies a given amount of bytes (that represents an element type) by the number of array elements. Anyway, the values used are kept under acceptable limits, according to what is defined in the WS benchmarks. All values used are summarized in Table 5.

All the attacks considered for this study were applied during invocations to the getInt and getString operations, with the exception of SOAP Array attack and Malicious Attachment that make sense only for the getArray and getFile operations, respectively. For the current study, we selected attack configuration values based on existing studies and security tools (Intel SOA Expressway, 2006; Jensen et al., 2009; "soapUI—Functional Testing," 2012; "WS-Attacker," 2012; "WS-Fuzzer Project," 2012; Suriadi et al., 2010). The specific values used are the ones presented in Table 1 and can be found in full detail in Oliveira et al. (2015).

Note that our goal is not to study the best attack configurations or to fine-tune the associated parameters. In fact, other values or combination of values are possible for configuring the attacks, however these common configurations serve our experimental goals, although giving an optimistic view of the behavior of frameworks when facing attacks. Thus, more specialized attacks can result in additional damage to the platforms and our tool and experimental setup can be easily used in such specific scenarios.

4.4. Executing and monitoring the tests

We applied our testing approach to test the seven service frameworks mentioned earlier, all deployed on top of Apache Tomcat. We used the following durations for each test run:

- **Pre-attack phase:** warm-up period (5 min); normal period (5 min);
- **Attack phase:** 15 min;
- **Post-attack phase:** normal period (5 min); rest period (5 min).

The above can be configured to different values, depending on the specificities of the platforms being tested, testing environment, or experimental goals. However, these values, empirically defined, are sufficient to disclose security issues, and should be kept practical, since using high durations is usually not an option for developers that frequently have time limits for testing tasks. In some cases, these durations are not enough to fully understand the behavior of the service platform that, being affected by an attack (displaying a dubious behavior, such as high memory use), presents no clear indication of a failure. Therefore, in such cases, we extended the duration of the rest period to 1 h. The goal is to detect if the dubious behavior (e.g., high memory use) remains or if the platform returns to normal patterns with, for instance, garbage collector calls decreasing high values of allocated memory to regular levels.

Client requests were generated and sent in a synchronous non-parallel fashion every 7 s after receiving each response (Ranjan et al., 2009). The timeout value of each request was set to 1 h to detect the cases where the web service does not provide a timely response. This high value gives us more confidence that a response will in fact not be received. All experiments that resulted in a dubious behavior of the service platform were repeated three times, with the goal of verifying possible deviations. During these experiments no significant deviations to the observed behavior in the first run were found (detailed information regarding all tests is available at Oliveira et al. (2015)).

During the tests, the server platform was continuously monitored using JConsole 1.6.0_30-b12. Based on previous studies (Gang et al., 2006; Suriadi et al., 2010), we opted to observe the following parameters: Java virtual machine memory heap size, number of allocated threads, and CPU usage. These can provide important information regarding the behavior of a given system in this kind of environments (Gang et al., 2006; Suriadi et al., 2010) and it is likely that there is an observable variation of these parameters when an insecure system is processing malicious requests.

Note that the decisions presented above respect to the experimental setup and are an instantiation of the approach to a specific Java-based environment. Due to the generality and flexibility of the proposed approach, web service providers or developers can to instantiate it to suit the needs of other specific environments (e.g., .NET based systems, Python web services).

Table 6
Summary of the problems detected.

Framework	Failures					Dubious behavior
	C	R	A	S	H	
Apache CXF						
Metro						1
Axis 2			1			1
Axis 1			2			2
Spring JAX-WS			1	1		
Spring-WS			1			2
XINS			2			1

5. Results

Table 6 presents the classification of the observed failures, using the CRASH scale (see Section 3.4), and includes a count of dubious behaviors (behaviors that do not represent clear failures) observed for the latest versions the frameworks tested. As we can see, five out of the seven frameworks presented at least one type of failure, which in practice means that services deployed using these frameworks may be vulnerable to security attacks. We also observed dubious behaviors in five of the seven frameworks.

Table 7 presents an overview of the results from the attack perspective, showing the total number of failures and dubious behaviors detected for each different attack. Coercive Parsing and Malicious Attachment attacks are the ones that lead to more failures in the frameworks under test. The Soap Array attack type was the one that allowed uncovering more dubious behaviors. In total, six out of nine types of attacks caused some kind of failure in the frameworks tested. Note that these attacks have been known for several years now and, as such, existing frameworks should provide some form of protection against them.

Table 7
Detected problems grouped by attack.

Attack	Occurrence	
	Failures	Dubious
Coercive Parsing	2	1
Malformed XML	1	–
Malicious Attach	2	1
Oversized XML	1	1
Repetitive Entity Expansion	–	–
Soap Array attack	1	3
XML Bomb	1	–
XML Document size	–	1
XML External Entities	–	–

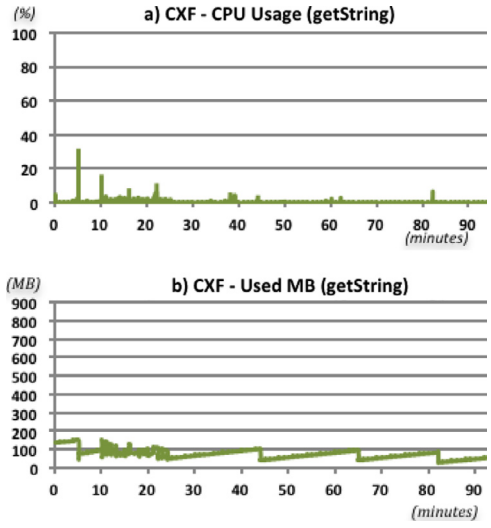

Fig. 4. CXF: Oversized XML attack.

Although we detected variations in CPU usage and memory allocation during the experiments, we never observed any relevant issues regarding the number of live threads or service response times. Due to this fact, these parameters are not considered in the following discussion, but can be consulted in the complete online datasets available at Oliveira et al. (2015). The next sections present an overview of the behavior of each tested framework, explaining the observed failures and dubious behaviors in detail.

5.1. Apache CXF

No failures or dubious behaviors were observed in Apache CXF. As an example, Fig. 4 presents a test run using the Oversized XML attack against the getString operation. As we can see, although the overall behavior changes when the attack is started, with variations in CPU use (Fig. 4a) and memory allocation (Fig. 4b), there is no perceptible increase in these parameters (Fig. 5).

5.2. Oracle Metro

We did not observe any failure in the latest version of Metro, however a potentially critical **dubious behavior** was identified. When a malicious attachment is sent to the service, such as the 100MB file in the Malicious Attachment attack, the framework creates a replica of the file in a server directory and it repeats the procedure if another identical attack is sent (which does not occur with a regular sized attachment). Although this is not a failure on its own, it is easy to note that, unless there is a mechanism that automatically deletes those particular files (and that mechanism has to be certain that the files are not needed), this can eventually fill up the storage space,

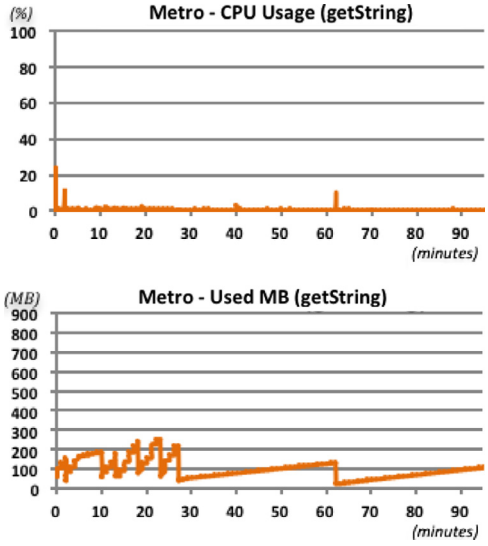

Fig. 5. Metro: Oversized XML attack.

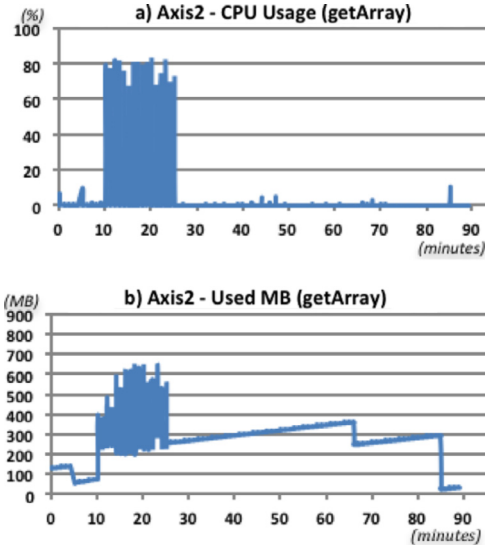

Fig. 6. Axis 2: Soap Array attack.

potentially damaging the regular service operation (which typically uses disk based-services, such as a database), or other disk-dependent applications executing in the same machine, ultimately affecting the regular behavior of the operating system.

5.3. Apache Axis 2

One failure was observed in Axis 2. During the execution of the Coercive Parsing attack, the CPU usage reached nearly 50% (which occurs for both the getInt or getString operations) and the server continuously logged a message that indicated that an error occurred in *org.apache.axiom.om.impl.llom.OMElementImpl. findNamespaceURI(OMElementImpl.java:497)*. Moreover, the client received consecutive responses with a *java.lang.StackOverflowError* faultstring, confirming that there was an internal failure while handling the request. We classified this behavior as an **Abort failure.**

A **dubious behavior** was detected when executing the tests on the Axis 2 framework. The Soap Array attack led this framework to consume in average 400MB of memory during the attack period (see Fig. 6b), which is twice the amount observed during the same attack in Metro and CXF (not visible in the figures, but available at Oliveira et al. (2015)). Also, Axis 2 raised the CPU usage frequently up to 80%

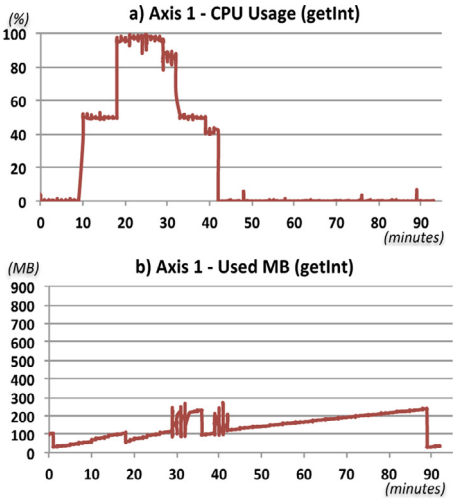

Fig. 7. Axis1: Coercive Parsing attack.

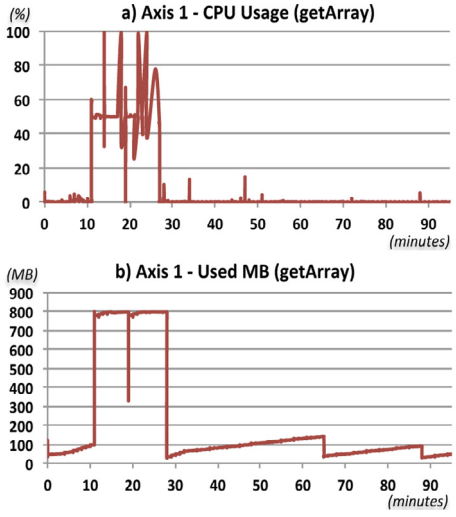

Fig. 8. Axis 1: Soap Array attack.

during the attack, approximately five times more than the peak values that were observed in Metro and CXF that consumed about 15% of CPU in the same period. Despite the observed behavior for Axis 2, the framework was able to recover to normal values around 1 h after starting the rest period. As we will see in the following section, the previous version of Axis shows even greater difficulties when handling this attack, essentially doubling the amount of memory required, using more time to reply to the client and becoming unresponsive during the attack period.

5.4. Apache Axis 1

Axis 1 presented two failures and one dubious behavior. Fig. 7 represents the behavior of Axis 1, when facing the Coercive Parsing attack targeting the getInt operation. As we can see, the CPU usage reaches high values, touching around 50% during the attacks and 100% in the 10 min that immediately follow the attack phase (this also occurs with the getString operation). During these 10 min there is a continuous output of a *StackOverflowException* to the server logs, which is an indicator of the occurrence of an internal error, and was classified as an **Abort failure.**

We observed another **Abort failure** in Axis 1 when executing the Soap Array attack (see Fig. 8). Shortly after sending the first attack, the allocated memory rises and maintains itself close to 750MB, with

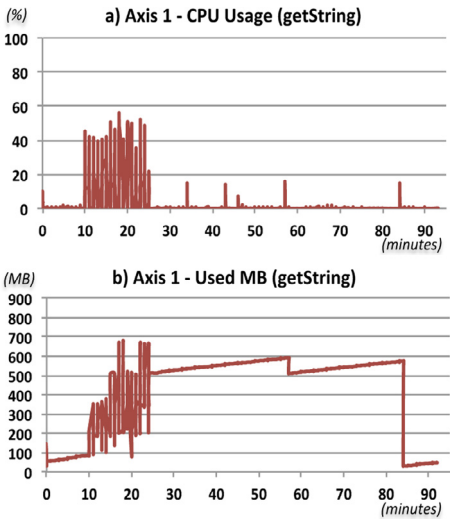

Fig. 9. Axis 1: Oversized XML attack.

the CPU achieving an approximate average usage of 50%, with sporadic peaks reaching 100%. In the meantime, the WSFAggressor client remains idle, waiting for a response and the Tomcat logs report the occurrence of *OutOfMemory* exceptions, with an indication of the failure of internal Tomcat components. After about 8 min an *OutOfMemory* exception is finally delivered to the client that issues then another attack. The same behavior is observed and is followed by a steep decrease of both CPU and memory usage (short after the start of the Post-Attack phase).

During the abovementioned anomalous periods, we tried to check if the server was still responsive and thus issued regular requests to the Axis service with an external tool, with no response being obtained. Furthermore, we tried to confirm if this failure affected other services in Tomcat. For this we issued a new request to another framework (simultaneously deployed in Tomcat), with no response being obtained (although the container should provide isolation among applications, that is not the case). We finally performed an extra verification to check if Tomcat's web page service engine could still serve HTTP requests (we issued a request to the Tomcat's default webpage), which was also not the case. However, despite such unresponsiveness periods and failures, we observed that the platform was able to recover after dealing with each attack during 8 min.

Finally, when executing the tests with the Oversized XML and XML Document Size attacks we detected a dubious behavior reflected by high CPU usage and memory allocation. As we can see in Fig. 9, the CPU usage increases during the attack period to about 40% and there is also an increase of the allocated memory in this period to nearly 500MB. Also, the allocated memory remains close to this level during the Post-attack phase. To better understand the problem, we extended the Keep period with an observation period of 1 h. During this time (i.e., the post-attack phase) the CPU usage remained close to zero with small sporadic usage peaks. However, we verified that the memory continues allocated, being released only at the end of the observation period (i.e., after almost 1 h). This is far from being an ideal behavior since it diminishes the resources available for the framework and other applications deployed in the server.

5.5. Spring-WS

We encountered one failure and two dubious behaviors in Spring-WS. Fig. 10 presents the behavior of Spring-WS in the presence of the Malicious Attachment attack, which is executed against the getFile Operation. In particular, Fig. 10(b) shows that, within 1 min after the first request with a SOAP attachment is received, the framework

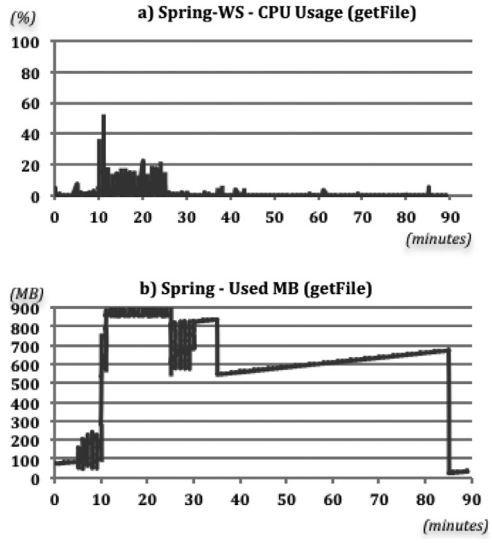

Fig. 10. Spring-WS: Malicious Attachment attack.

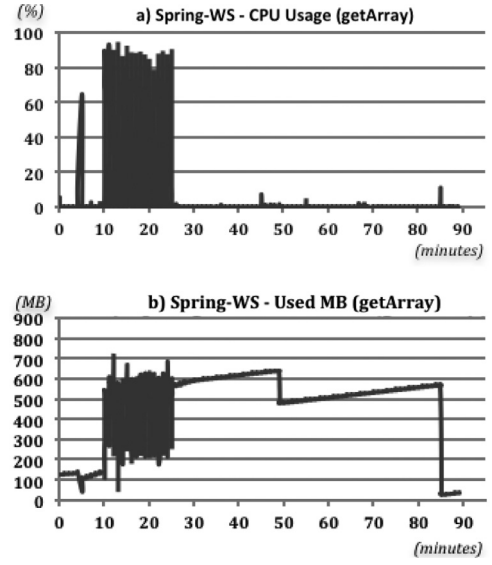

Fig. 11. Spring-WS: SOAP Array attack.

reaches 900MB of allocated memory, and CPU usage increases up to approximately 55%. Consequently, a *java.lang.OutOfMemoryError: Java heap space* error message is returned to the client and for each malicious request sent afterward, the framework returns the same exception as a response.

During the execution of the attack, we tried to confirm if the server was still responsive and thus issued regular requests to the Spring-WS service with an external tool, which confirmed that the service was able to respond accordingly (contrarily to the failure observed in Axis 1 during the Soap Array attack with the same error message). We also observed that the services that were available on other frameworks installed in the same server were also able to respond, and the application server (Tomcat) administration web page was available. The framework was able to recover to normal memory values approximately 1 h after starting the rest period. The unexpected exception that resulted from consuming most of the memory allocated for the framework qualifies as an **Abort failure.**

The first of the two dubious behaviors detected occurred when the Soap Array attack was executed. As we can see in Fig. 11(a), this attack led the framework to use between 80% and 90% of CPU. In terms

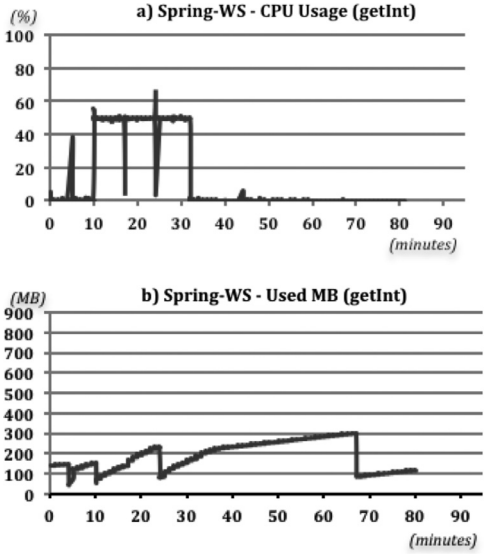

Fig. 12. Spring-WS: Coercive Parsing attack.

of memory, the attack caused the framework to consume consistently 600MB of memory during the attack period (see Fig. 11b), with sporadic 700MB peaks. Despite this behavior, the framework was able to recover to normal values about 1 h after the beginning of the rest period.

The second dubious behavior was found during the attack phase, when the Coercive Parsing attack was being executed. Spring-WS was only able to process three requests from the attackload, taking approximately 7 min to handle each one (extending the attack phase from 15 to approximately 21 min). We conducted some tests to if the framework was able to respond to concurrent requests while the attack was being processed and found out that this behavior did not have any impact on other clients (i.e., the service was not affected). Despite we did not observe any abnormal memory usage while the framework was processing the attacks, it used frequently up to 50% of the CPU (see Fig. 12a).

A close inspection of the Tomcat logs revealed that, after processing each request, Spring-WS logged the following error message *SAAJ0511: unable to create envelope from given source*, and that WSFAggressor received the following response: *the request sent by the client was syntactically incorrect ()*. Although the framework response is acceptable and we did not found a real indication of an internal error (e.g., an unexpected exception) we consider that the amount of CPU used by the framework in these circumstances and its inability to process more than three sequential malicious requests during 21 min are not acceptable behaviors.

5.6. Spring JAX-WS

The typical behavior of Spring JAX-WS when handling the Malformed XML attack is shown in Fig. 13. Although no perceptible deviations can be seen (the same happens for the other attack types), two failures were raised. The first was an **Abort failure** when executing the Malformed XML attack. In this case, a *javax.xml.bind.UnmarshalException* was thrown and delivered to the client. This exception includes a reference to a *WstxParsingException*, raised by the XML parser used by Spring-WS (Woodstox), which is related to an unexpected closure of an XML tag. We investigated this behavior in the server logs and discovered that a *NullPointerException* was also raised during the attacks (and wrapped in the *UnmarshalException*), indicating the incapability of the framework to handle an unexpected case.

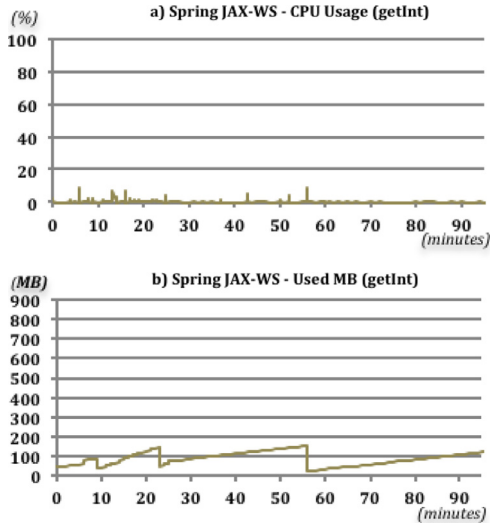

Fig. 13. Spring JAX-WS: Malformed XML attack.

The second failure observed was a **Silent failure:** after launching the first request of the Oversized XML attack, the client did not obtain a response from the server until the end of the Keep period. Therefore, we extended the Keep period to 1 h to verify if the service platform could still (although late) deliver a response to the client, which did not occur. This was classified as a Silent failure since the operation did not conclude and the server indicated no error. Despite this, we verified that the platform was still able to respond, during this time, to other regular requests (submitted in parallel), which means that it was not blocked to all requests (or otherwise this could be classified as a more severe Restart failure).

Although in some cases it can be acceptable that a framework ignores an attack, the behavior described above seems to indicate that some internal problem has occurred, since the framework did not log or report the occurrence of an anomalous situation (the identification of a suspicious request, or the eventual countermeasure taken that results in no response being delivered to the client). We then tried to see what could be happening during that period and used "Wireshark" (2012), a TCP eavesdropping tool, to understand if the framework received the attack or not. Based on the information collected by Wireshark, we realized that, although the server platform received the attack, it started to continuously reply with TCP Zero Window packets, which essentially indicates a resource issue in the receiver, as the application is not retrieving data from the TCP buffer in a timely manner ("TCP Analyze Sequence Numbers—The Wireshark Wiki," 2012). This behavior was observed until the end of the experiment, and confirms the framework's inability to handle this attack.

5.7. XINS

Two failures and a dubious behavior were observed in XINS. Fig. 14 presents the CPU and memory used by XINS when processing the Malicious Attachment attack. As we can see, XINS is not particularly optimized to handle SOAP attachments as it allocates nearly 300MB to handle a normal 700KB file (twice as much as Apache CXF). When a 100MB file is sent in the attack phase the CPU increases to nearly 50% and the allocated memory reaches almost 800MB. XINS logs an *OutOfMemoryError* for each request received (the client receives an *InternalError* message) and becomes unable to process parallel requests (the Tomcat administration console also becomes unavailable after the occurrence of that exception). This was classified as an **Abort failure.**

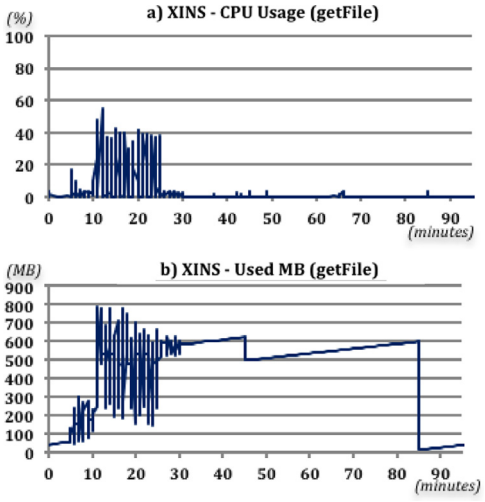

Fig. 14. XINS: Malicious Attachment attack.

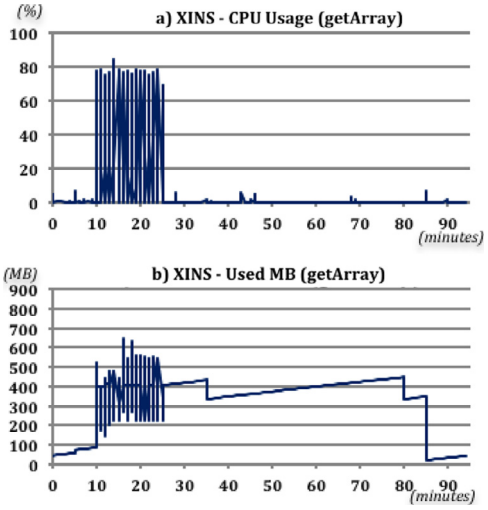

Fig. 15. XINS: Soap Array attack.

The second failure detected was also an **Abort failure,** and was observed when executing the XML Bomb attack (see Fig. 16). At the beginning attack phase the allocated memory increased up to 800MB, ultimately resulting in an *OutOfMemoryError* exception being thrown by the server. After this, it was not possible to continue monitoring the execution of the attack phase (and remaining phases) using JConsole, since this tool also crashed as a result of the server behavior. However, using the Windows task manager we could observe that the Tomcat process kept the allocated memory at 887MB after the first exception, and reached 1.38GB when the second exception occurred (memory allocation oscillated between these two values during the entire attack phase). We also observed that XINS was able to deliver responses to non-malicious requests during the attack phase (i.e., the server was accepting requests), although those responses included an *InternalError* message indicating the framework's internal failure. After the conclusion of the attack phase, XINS was again able to return valid messages in response to non-malicious requests.

Finally, we observed a **dubious behavior** while executing the Soap Array attack. As we can see in Fig. 15 (a), during the attack phase, the CPU usage increased up to 80%, which is a large increase when considering, for example, the behavior of Apache CXF in the same situation. During this period, the allocated memory reached approximately 650MB and it took more than 1 h for the framework to release the memory, as shown in Fig. 15(b).

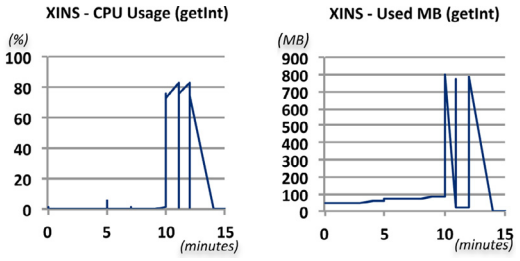

Fig. 16. XINS: XML Bomb attack.

6. Further discussion

In this section we present a general discussion of the impact of the attacks in the frameworks in a quantitative manner. Also, in order to understand evolution of the security characteristics of the frameworks over time, we compare the failures modes and dubious behaviors observed in two different versions of three of the frameworks in the presence of DoS attacks.

6.1. Analyzing the impact of the attacks

For better understanding the impact of DoS attacks on the tested frameworks, and since in some cases it can be quite difficult to assess whether an attack actually impacts the system under testing (e.g., due to small or non-perceptible variations of the parameters being observed), we measured the total CPU usage and allocated memory for each period of the tests (i.e., Warm-up, Normal, Attack, and Rest). These measurements correspond to the respective areas of the graphs shown in the previous section. The area for a given period P can be determined using the following formula:

$$\text{Area}\, P = \int_a^b f(t)dt \tag{1}$$

In formula (1) a and b correspond to the start time and finish time of a given period P. For instance, for the Warm-up period in our experiments, these two values are 0 and 5, respectively.

After calculating the eight areas for each framework tested (four periods per each of the two two-system parameters, CPU and memory), we determined the relative change (RC) (Bennett and Briggs, 2010) between three key pairs of periods: *Normal and Attack {N, A}; Normal and Regular {N, R}*, and *Warm-up and Keep {W, K}*. The relative change describes the difference between a reference value and a new one, thus being suitable for understanding the impact of the attacks, in this case, in terms of changes in the system parameters (Bennett and Briggs, 2010). For a given pair of periods {P1, P2}, the relative change (RC) is calculated using:

$$\text{RC}\{\text{P1}, \text{P2}\} = \frac{\text{Area P2} - \text{Area P1}}{\text{Area P1}} \tag{2}$$

Relative change and special cases of it like relative error (i.e., quantifies an error ratio between a true and a measured value using the same formula as RC) (Abramowitz and Stegun, 1965), are widely accepted in the research community and are applied in different areas like signal and image processing (Zhang and Yang, 2012).

Table 8 presents the relative change values (rounded to the units), grouped by framework. For each framework we show the results for the three key pairs of periods ({N, A}, {N, R}, and {W, K}) and then by CPU and memory. For presentation clarity, we do not show any relative change values inferior to 1. In addition, we highlighted in color the top seven values found for all frameworks, in each of the six different cases: 1) {N,A} for CPU; 2) {N,A} for memory; 3) {N,R} for CPU; 4) {N,R} for Memory; 5) {W,K} for CPU; and 6) {W,K} for Memory.

Considering the **overall attacks results,** we can see that the Soap Array attack has some impact on four out of the seven frameworks (the exceptions are CXF, Metro and Spring JAX-WS, which are not present at the top values). The attack results in particularly high RC values for CPU and memory usage in Axis 1, Axis 2, Spring-WS and XINS, especially for {N, A}. Concerning the impact of this attack in {N, R} we did not detect any significant changes in the memory usage. Clearly the handling and processing of soap arrays is an aspect that the developers of these frameworks must improve.

The Coercive Parsing attack also impacts three frameworks: Axis 1, Axis 2, and Spring-WS. Axis 1 presented the highest value observed during the experiments (an RC value of 1749), while Axis 2 and Spring-WS showed lower but still considerably high RC Values (748 and 711, respectively). Developers and service providers need to understand the impact of these two attacks and additional protection measures should be put in place when using these frameworks.

Overall, we can also see that the XML External Entities and Repetitive Entity Expansion attacks are the ones that cause fewer problems to the frameworks, which, in general, appear to have the mechanisms needed to adequately handle this kind of requests.

Concerning the individual **frameworks results,** Axis 1 shows the greatest number of top issues, including potentially severe behaviors that manifest even after the regular period has finished (i.e., relative to the RC{W, K} values). This is the case of the results obtained for the Coercive Parsing, Oversized XML, and Document Size attacks. In addition to the impact of the Soap Array attack mentioned before, the RC{N, A} for CPU usage in Axis 2 is also quite high for the Coercive Parsing attack, an aspect that should be handled properly by such a popular framework.

Spring-WS also presents very high values in terms of RC{N, A}, namely in what regards: i) CPU values when executing the Coercive Parsing attack, ii) the memory values during Malicious Attachment attacks, and iii) the CPU and memory values during the SOAP Array attack. The RC{W, K} value for memory consumption after executing Malicious Attachment attacks is also significant and shows the difficulty of the framework in releasing the allocated memory. These values emphasize the observations presented in Section 5.6 and are a topic of particular concern as Spring-WS is actively maintained (which is not the case of Axis 1, for example).

As mentioned earlier, the XINS framework has difficulties in handling the Malicious Attachment attack. Such difficulties remain for more than 1 h after the end of the attack phase, which translates into one of the top detected memory issues for RC{W, R} (holding a final value of 11). The remaining values, in particular the CPU usage RC values, are not in the top issues, but that is simply because the framework also has difficulties handling the regular size attachments (as mentioned in the previous section). This decreases the overall RC value, but does not remove the problem detected before.

An interesting aspect that it is worth mentioning is that there seems to be some relation between the XML Parser used by the frameworks and the existence of failures and dubious behaviors. Axis 1, Spring-WS and XINS used the XML parser Xerces and were the frameworks more prone to allocate significant system resources. In fact, the tests performed uncovered an *OutOfMemoryException* in these three frameworks (although with different attacks), which suggests that Xerces might be more vulnerable to brute force attacks. On the other hand, CXF, Metro and Spring JAX-WS were the most resource efficient frameworks, and the three used the Woodstox XML Parser. The difference in terms of the XML Parsers might also be the reason why Spring-WS and Spring JAX-WS presented so different behaviors during our security tests. Anyway, the impact that XML parser and other internal components of the frameworks have in the security is a topic that needs be further explored in future studies.

Table 8
Relative change values.

	Axis 1						Spring JAX-WS						Spring WS						XINS					
	{N,A}		{N,R}		{W,K}		{N,A}		{N,R}		{W,K}		{N,A}		{N,R}		{W,K}		{N,A}		{N,R}		{W,K}	
Attack	CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM
CP	1749	5	-	-	35	3	-	10	-	4	2	6	711	3	-	1	-	1	16	2	-	-	-	1
MX	31	6	-	-	-	-	9	3	-	-	-	2	3	3	-	-	-	-	2	3	-	1	-	1
MA	7	4	16	1	-	-	34	3	-	-	-	-	17	21	-	4	1	8	18	9	-	2	-	11
OX	163	14	1	-	-	10	-	3	-	-	-	1	17	5	-	1	-	1	8	1	-	-	-	-
REE	1	4	5	-	-	1	2	2	-	-	-	2	1	2	-	-	-	-	2	2	-	-	-	1
SA	551	38	1	1	-	1	133	7	-	1	-	2	415	12	-	4	-	4	273	16	-	4	-	7
XB	3	3	1	-	-	3	8	3	-	-	-	-	5	2	-	1	-	--	-	-	-	-	-	-
XDS	278	21	1	-	-	9	3	3	-	-	-	1	48	4	-	1	-	1	25	4	-	-	-	-
XEE	19	4	-	-	-	1	3	3	-	-	-	2	3	3	-	-	-	-	5	3	-	-	-	1

	Axis 2						CXF						Metro					
	{N,A}		{N,R}		{W,K}		{N,A}		{N,R}		{W,K}		{N,A}		{N,R}		{W,K}	
Attack	CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM	CPU	MEM
CP	748	-	-	-	-	-	1	-	-	-	-	-	37	-	-	2	-	2
MX	8	5	--	-	-	-	9	3	-	-	-	-	16	5	2	-	1	-
MA	33	7	-	1	-	-	32	3	-	-	-	-	43	2	-	-	-	-
OX	12	3	-	-	-	-	6	2	-	-	-	-	2	6	-	1	1	-
REE	8	4	-	1	-	-	3	4	-	-	-	-	2	4	-	-	-	-
SA	495	20	-	4	-	9	6	4	-	1	1	1	49	9	-	1	-	1
XB	5	5	-	1	-	-	6	4	-	1		-	2	4	-	-	1	-
XDS	32	6	-	1	-	1	6	4	-	-	-	-	20	4	-	-	-	-
XEE	18	5	-	2	-	1	14	3	-	-	-	-	7	5	--	2	-	-

6.2. Analyzing the evolution across versions

To study the evolution of the frameworks over time we also tested a previous version of three of the frameworks: Axis 2, CXF and Metro (see Table 3). In practice, the goal is to compare the failures modes and dubious behaviors observed in the two different versions of each of three of the frameworks in the presence of DoS attacks. These frameworks were selected due to their prevalence in real installations.

In general, we did not found considerable improvements when comparing the latest versions with the older ones. In fact, regarding **Axis 2** it was quite the opposite: for example, we observed a significant increase in the CPU usage (from 60% in version 1.6.1 to 80% in version 1.6.2) in the presence of the SOAP Array attack. Also, the Abort failure observed in the latest version of Axis 2 during the execution of the Coercive Parsing attack (in the form of a *java.lang.StackOverflow* exception) was not observed in the older version.

When comparing versions 2.6.1 and 3.0.3 of **Apache CXF** we observed a behavior that we consider as an improvement in the way SOAP Array attacks are handled. While the latest version logs the error message *Unmarshalling Error: Maximum Number of Child Elements limit (50,000) Exceeded* when this attack is performed, in the older version the expected result is returned (i.e. the size of the array) after a few seconds with a minimal impact on the server resources (i.e. the CPU usage remained consistently below 10% while the memory allocation never exceeded 250MB). From the security perspective, we consider that this validation/limitation of the size of the array in version 3.0.3 is an important security improvement as it becomes more difficult for the malicious users to take advantage of extremely large arrays to cause a DoS.

A clear improvement was observed when comparing versions 2.1.1 and 2.3.1 of **Metro.** A Silent failure was observed in the older version after an Oversized XML attack was issued. In practice, although the framework was able to process requests from concurrent clients, it became unable to respond to the client that issued the attack and remained in that state until the end of the rest period. In the latest version, the framework aborts the execution of the malicious request about 3 min after the attack and issues a *ClientAbortException* that is recorded in the Tomcat logs. The client receives a *java.net.SocketException: connection reset error* and the framework returns to a state in which it is again able to accept requests from the same client. This indicates that Metro 2.3.1 tries to process the received requests during a given period of time and aborts the execution after a timeout. Although 3 min may not be an adequate value for all cases, one can consider that this corresponds to a more adequate behavior.

There is still one problem that needs to be addressed urgently in Metro. After the experiments were concluded, we noticed that Metro 2.3.1 created a temporary file on the remote machine for each Malicious Attachment attack that was sent (i.e. a request with a 100MB file as attachment). In particular, Metro 2.3.1 was able to handle a combined payload of 300MB per min (i.e. three files were stored in the server disk per minute). Most importantly, the temporary files remained at the server and were not deleted by Metro, after the attack. Malicious users can explore this issue and send a very large number of requests with hundreds of megabytes in attachments and force that the operating system to slow down or even crash (due to a full disk). Version 2.1.1 had the exact same problem as version 2.3.1, which means that this problem is not new and has persisted during different versions of Metro. If it is not addressed it can bring catastrophic consequences to service providers.

7. Conclusion

In this paper we studied the behavior of well-known web service frameworks in the presence of DoS attacks. Using related security research works and web service attacking tools as basis, we compiled and executed a set of DoS attacks using the security testing tool WSFAggressor. We defined a multi-phase testing approach with the goal of observing the behavior of service platforms during attacks and detecting any possible attack effects during normal service or idle operation of the system. In practice, developers and providers can use the proposed approach to assess the security of service platforms.

Results indicate that web service frameworks are in general resistant to attacks, with Apache CXF and Oracle Metro displaying no failures at all. However, they also point out severe failures and dubious behaviors in the remaining frameworks, indicating that urgent improvements are required. Although only abort and silent failure modes were observed, we believe that all failure modes defined

by the CRASH scale are useful and might be observable when testing other frameworks, or using other types of attacks.

Building on top of the approach presented in this paper, future work should study the impact of web service attacks on the execution of genuine non-malicious operations. In other words, the proposed procedure should be extended to understand how much a given attack, executed against a framework, might affect a genuine client that is trying to invoke a service (supported by the same framework).

Acknowledgments

This work has been partially supported by the Project ICIS—Intelligent Computing in the Internet of Services, number 4774, of the program CENTRO-SCT-2011-02, and by the projects CErtification of CRItical Systems (CECRIS), Marie Curie Industry-Academia Partnerships and Pathways (IAPP) number 324334; and DEsign, Verification and VAlidation of large-scale, dynamic Service SystEmS (DEVASSES), Marie Curie International Research Staff Exchange Scheme (IRSES) number 612569, both within the context of the EU Seventh Framework Programme (FP7).

References

Abramowitz, M., Stegun, I.A. (Eds.), 1965. Handbook of Mathematical Functions: With Formulas, Graphs, and Mathematical Tables, 0009-Revised edition. Dover Publications, New York.

Antunes, N., Laranjeiro, N., Vieira, M., Madeira, H., 2009. Effective detection of SQL/XPath injection vulnerabilities in web services. In: IEEE International Conference on Services Computing, 2009. IEEE, pp. 260–267. doi:10.1109/SCC.2009.23.

Apache Axis2/Java [WWW Document], 2012. URL https://axis.apache.org/axis2/java/core/ (accessed 22.03.12).

Apache Axis [WWW Document], 2006. URL https://axis.apache.org/axis/.

Apache CXF [WWW Document], 2012. URL https://cxf.apache.org/.

Apache Tomcat [WWW Document], 2012. URL https://tomcat.apache.org/ (accessed 22.03.12).

Bennett, J.O., Briggs, W.L., 2010. Using and Understanding Mathematics: A Quantitative Reasoning Approach, fifth ed. Addison-Wesley.

Cloudfare, 2014. Affordable advanced DDoS protection and mitigation [WWW document]. URL https://www.cloudflare.com/ddos (accessed 03.03.15).

Comparing Java 2 EE and .NET Framework [WWW Document], 2004. URL http://java.sun.com/performance/reference/whitepapers/WS_Test-1_0.pdf.

Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N., Weerawarana, S., 2002. Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI. IEEE Internet Comput. 6, 86–93. doi:10.1109/4236.991449.

Defeating DDOS Attacks [WWW Document], 2014. Cisco. URL http://cisco.com/c/en/us/products/collateral/security/traffic-anomaly-detector-xt-5600a/prod_white_paper0900aecd8011e927.html (accessed 03.03.15).

Duchi, F., Antunes, N., Ceccarelli, A., Vella, G., Rossi, F., Bondavalli, A., 2014. Cost-effective testing for critical off-the-shelf services. In: Bondavalli, A., Ceccarelli, A., Ortmeier, F. (Eds.), Computer Safety, Reliability, and Security. Lecture Notes in Computer Science. Springer International Publishing, pp. 231–242.

Falkenberg, A., Mainka, C., Somorovsky, J., Schwenk, J., 2013. A new approach towards DoS penetration testing on web services. In: 2013 IEEE 20th International Conference on Web Services (ICWS), pp. 491–498. doi:10.1109/ICWS.2013.72.

Gang, W., Cheng, X., Ying, L., Ying, C., 2006. Analyzing XML parser memory characteristics: experiments towards improving web services performance. In: International Conference on Web Services, 2006. ICWS '06. IEEE, pp. 681–688. doi:10.1109/ICWS.2006.31.

Govindaraju, M., Slominski, A., Chiu, K., Liu, P., van Engelen, R., Lewis, M.J., 2004. Toward characterizing the performance of SOAP toolkits. In: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing, 2004. IEEE, pp. 365–372. doi:10.1109/GRID.2004.60.

Imperva, 2012. Report #12—Denial of Service Attacks: A Comprehensive Guide to Trends, Techniques and Technologies.

Intel SOA Expressway, 2006. Protecting Enterprise, SaaS & Cloud-based Applications—A Comprehensive Threat Model for REST, SOA and Web 2.0.

Jensen, M., Gruschka, N., Herkenhöner, R., 2009. A survey of attacks on web services. Comput. Sci. Res. Dev. doi:10.1007/s00450-009-0092-6.

Koopman, P., Sung, J., Dingman, C., Siewiorek, D., Marz, T., 1997. Comparing operating systems using robustness benchmarks. In: The Sixteenth Symposium on Reliable Distributed Systems, 1997. Proceedings. IEEE, pp. 72–79. doi:10.1109/RELDIS.1997.632800.

Mainka, C., Somorovsky, J., Schwenk, J., 2012. Penetration testing tool for web services security. In: 2012 IEEE Eighth World Congress on Services (SERVICES), pp. 163–170. doi:10.1109/SERVICES.2012.7.

McDowell, M., 2013. Understanding Denial-of-Service attacks | US-CERT [WWW document]. URL http://www.us-cert.gov/ncas/tips/st04-015 (accessed 25.02.14).

Metro [WWW Document], 2012. URL http://metro.java.net/.

Microsoft, 2008. Comparing .NET 3.5/Windows Server 2008 to IBM WebSphere 6.1/Red Hat Linux Web Service Performance.

Oliveira, R.A., Laranjeiro, N., Vieira, M., 2015. JSS'15 WSFAggressor tool and experimental data [WWW document]. URL http://eden.dei.uc.pt/~racoliv/papers/2015-jss-tool-exp-data.zip.

Oliveira, R.A., Laranjeiro, N., Vieira, M., 2012. Experimental evaluation of web service frameworks in the presence of security attacks. In: IEEE Ninth International Conference on Services Computing (SCC 2012). IEEE Computer Society, Honolulu, Hawaii, USA.

Oliveira, R.A., Laranjeiro, N., Vieira, M., 2012. WSFAggressor: an extensible web service framework attacking tool. In: Proceedings of the Industrial Track of the 13th ACM/IFIP/USENIX International Middleware Conference, MIDDLEWARE '12. ACM, New York, NY, USA, pp. 2:1–2:6. doi:10.1145/2405146.2405148.

Oracle, 2013. What are RESTful web services? [WWW document]. URL http://docs.oracle.com/javaee/6/tutorial/doc/gijqy.html (accessed 23.02.15).

Ranjan, S., Swaminathan, R., Uysal, M., Nucci, A., Knightly, E., 2009. DDoS-shield: DDoS-resilient scheduling to counter application layer attacks. IEEEACM Trans. Netw. 17, 26–39. doi:10.1109/TNET.2008.926503.

soapUI—Functional Testing [WWW Document], 2012. URL http://www.soapui.org/ (accessed 22.03.12).

Spring support for JAX-WS RI—Project Kenai [WWW Document], n.d. URL https://jax-ws-commons.java.net/spring/ (accessed 25.02.15).

Spring Web Services - Home [WWW Document], 2013. URL http://docs.spring.io/ (accessed 21.10.13).

Sun Microsystems Inc., 2010. jax-ws: JAX-WS reference implementation [WWW document]. URL https://jax-ws.dev.java.net/ (accessed 14.02.08).

Suriadi, S., Clark, A., Schmidt, D., 2010. Validating denial of service vulnerabilities in web services. In: Fourth International Conference on Network and System Security (NSS), 2010, pp. 175–182. doi:10.1109/NSS.2010.41.

TCP Analyze Sequence Numbers—The Wireshark Wiki [WWW Document], 2012. URL http://wiki.wireshark.org/TCP_Analyze_Sequence_Numbers (accessed 22.03.12).

Transaction Processing Performance Council, 2008. TPC Benchmark App. (Application Server). V1.3.

U, G., Rao, S., 2005. Develop web services with Axis2, part 1: deploy and consume simple web services using the Axis2 runtime [WWW document]. URL https://www.ibm.com/developerworks/opensource/library/ws-webaxis1/ (accessed 22.03.12).

Vieira, M., Antunes, N., Madeira, H., 2009. Using web security scanners to detect vulnerabilities in web services. In: IEEE/IFIP International Conference on Dependable Systems & Networks (DSN 2009). IEEE Computer Society, pp. 566–571. doi:10.1109/DSN.2009.5270294.

Vieira, M., Laranjeiro, N., Madeira, H., 2007. Assessing robustness of web-services infrastructures. In: 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2007). IEEE Computer Society, Edinburgh, United Kingdom, pp. 131–136. doi:10.1109/DSN.2007.16.

Wireshark [WWW Document], 2012. URL https://www.wireshark.org/ (accessed 22.03.12).

WS-Attacker [WWW Document], 2012. SourceForge. URL http://sourceforge.net/projects/ws-attacker/ (accessed 22.03.12).

WSBang [WWW Document], 2012. URL http://www.isecpartners.com/application-security-tools/wsbang.html (accessed 22.03.12).

WSFuzzer Project [WWW Document], 2012. URL https://www.owasp.org/index.php/Category:OWASP_WSFuzzer_Project (accessed 22.03.12).

XINS—Open Source Web Services Framework [WWW Document], 2013. URL http://xins.sourceforge.net/ (accessed 21.10.13).

Zeichick, A., 2008. Tomcat, Eclipse named the most popular in SDTimes study [WWW document]. URL http://www.sdtimes.com/link/31882 (accessed 3.7.12).

Zhang, A., Yang, P., 2012. An improved algorithm for fractal image encoding based on relative error. In: 2012 Fifth International Congress on Image and Signal Processing (CISP), pp. 254–257. doi:10.1109/CISP.2012.6469909.

Rui Oliveira is a Ph.D. student at the University of Coimbra, Portugal, where he has been a researcher for about 5 years. His research interests include experimental security and performance evaluation in distributed systems, ranging from service oriented middleware to emergent and scalable cloud data services. He has authored four publications in leading peer-reviewed conferences in the services computing and dependability areas and has also served as reviewer for international workshops.

Nuno Laranjeiro received the Ph.D. degree from the University of Coimbra, Portugal, in September 2012, where he currently is an assistant professor. His research focuses on robust software services, including experimental dependability evaluation, web services interoperability and security and enterprise application integration. He has authored more than 40 peer-reviewed publications, including papers in refereed conferences and journals in the services computing and dependability areas. He has participated in several national and European research projects and served as program committee member for several international conferences and as reviewer for journals in the dependability and services areas.

Marco Vieira is an assistant professor at the University of Coimbra, Portugal. He is an expert on dependability benchmarking and his research interests also include experimental dependability evaluation, fault injection, security benchmarking, intrusion detection, software development processes, and software quality assurance, subjects in which he has authored or co-authored more than 120 papers in refereed conferences and journals. He has participated in many research projects, both at national and European level. He has served on program committees of the major conferences of the dependability area and acted as referee for many international conferences and journals in the dependability and databases areas.