WSFAggressor: An Extensible Web Service Framework Attacking Tool

Rui André Oliveira
CISUC/DEI, University of Coimbra
Coimbra, Portugal
racoliv@dei.uc.pt

Nuno Laranjeiro
CISUC/DEI, University of Coimbra
Coimbra, Portugal
cnl@dei.uc.pt

Marco Vieira
CISUC/DEI, University of Coimbra
Coimbra, Portugal
mvieira@dei.uc.pt

ABSTRACT

This paper presents a tool for testing the security of web service frameworks. The tool implements a large set of attack types, defined based on previous security research studies, existing testing tools, and field experience. The motivation is that developers frequently build web services based on the assumption that the underlying frameworks are secure, which is not always the case. Despite the evident need for security in the platforms that support services, existing security testing tools are very limited. In practice, most tools focus on application level vulnerabilities, and the few that allow testing platforms implement a very limited set of attack types. To the best of our knowledge, our tool includes more attacks than any other existing tool. Furthermore, by implementing an extensible architecture (based on plug-ins), the tool can be easily extended with additional attacks, supporting also a large variety of testing configurations. Results show that it can be used to disclose critical security problems in well-known frameworks.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging – *Testing tools.*

General Terms

Security, Experimentation, Verification, Performance, Design.

Keywords

Web Services, Web Service Frameworks, Security Testing.

1. INTRODUCTION

Web services (WS) are becoming increasingly used not only inside corporate environments, but also for interconnecting systems worldwide. They are self-describing components that can be invoked by other software applications across the web in a platform-independent manner, and are supported by standard protocols like SOAP and WSDL [1]. Deploying a web service requires, in addition to the actual service application, a set of other software components. In particular, we need a web services framework or stack and a server (typically a web server) to act as a container for the web service application. In this combined setup, depicted in Figure 1, the server delivers SOAP messages (arriving from clients via HTTP) to the framework, which then processes (parses the message, checks for errors, and builds a programming language-level object) and delivers those messages to the actual service implementation (i.e., the application) [2].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Middleware 2012 Industry Track, December 3–7, 2012, Montreal, Quebec, Canada.
Copyright 2012 ACM 978-1-4503-1613-2/12/12 …$15.00.

In practice, the WS framework provides core functionalities, being responsible for dealing with all client requests, including valid requests (compliant with the WS specification), invalid requests (not compliant with the WS specification), and malicious requests (that try to exploit existing vulnerabilities in the WS stack and, from a domain point-of-view, can be either valid or invalid).

Currently there are tens of web services frameworks available (e.g., Apache Axis 2, Metro, Apache CXF [3–5]). These frameworks apply well-known and largely tested technology that has been used for many years in corporate environments by millions of web service clients. However, contrary to what vendors and providers frequently advocate, such frameworks are not more secure than any other network-based systems, being largely exposed to vulnerabilities and also attacks originating from common Internet protocols (i.e., unspecific to the web services technology) [6–8]. In fact, previous research describes malicious payloads (attacks) that take advantage of framework security vulnerabilities. Most are based on specially-crafted requests, including malformed XML elements [6], [8]. However, performing an attack does not necessary require a malformed request to be sent. An attacker can build a valid and well-formed request that still maliciously exploits an existing vulnerability. For instance, an attacker can build large (yet valid) requests that may simply force the WS Framework to waste CPU cycles and/or memory, which can ultimately result in Denial of Service.

Software vendors are increasingly becoming aware of security issues in web services, as can be seen by the large number of vulnerability detection tools available nowadays. However, most of those tools focus on the detection of vulnerabilities at the application level (e.g., SQL Injection, XSS [9], [10]), disregarding the underlying framework, which is frequently (wrongly) assumed to be secure. The few tools that allow testing WS frameworks are very limited, as they implement only a small set of the relevant attacks, allow little (or no) configuration and tuning, and lack flexibility (and the required documentation) to be easily extended.

This paper presents WSFAggressor (freely available at [11]), an extensible and configurable tool that can be used to perform security testing on web services frameworks. The tool includes a set of nine attack types (defined based on previous security research studies, existing testing tools, and field experience) and can be

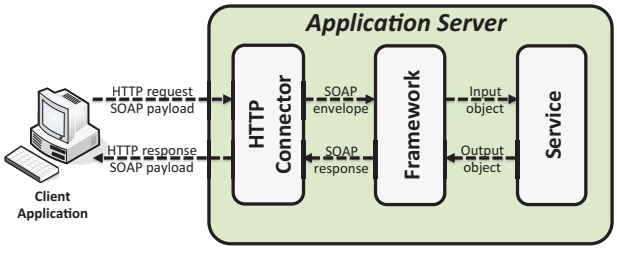


Figure 1. Web services supporting infrastructure.

easily extended by integrating plug-ins implementing other attacks. Also, it is highly configurable, supporting different testing profiles and attack combinations. Thus, it overcomes some of the limitations of current tools, enabling developers and providers to implement tailored testing approaches and attacks on top of an extensible software architecture. The usefulness of WSFAggressor is demonstrated by the results obtained when testing a simple web service application on top of Apache Axis 2, a widely used WS framework, and Axis 1, a well-known and matured framework. Results show that, in general, the frameworks are resistant to the security attacks executed by our tool. Despite this, we were able to expose major security failures and abnormal behaviors that require attention from developers and corrective measures so that the security of these frameworks can be improved. Note however that, the focus of the present work is to present the tool, the results presented are a subset of a larger evaluation performed in [17] and are included here to illustrate WSFAggressor's usefulness.

The structure of the paper is as follows. The next section describes our tool. Section 3 presents its architecture and Section 4 discusses the experimental results. Section 5 concludes the paper.

2. THE WSFAGGRESSOR APPLICATION

This section introduces the main features provided by WSFAggressor (available at [11]). This Java tool, built based on WS-Attacker [12], provides an easy to use interface for security testing of WS frameworks. Note that, the user interface is essentially the same as the one found in WS-Attacker. WSFAggressor's features do not focus on the user interface, however we include screenshots to provide an easier-to-follow explanation of its capabilities. The section concludes by positioning WSFAggressor among its main competitors: SoapUI and WSFuzzer.

2.1 Selecting and Executing Attacks

The application's user interface is organized in a set of tabs that group and separate the main operations (see Figure 2 for an overview of the interface). After the application is launched the 'WSDL Loader' tab is selected by default and provides options for retrieving information regarding the target web service (i.e., its interface). The user simply needs to enter the URL location of the WSDL file and the application retrieves the interfaces and operations available from the web service. The user can then select the operation that will be the target of the security test and can also visualize the required operation input parameters [11].

Figure 2 presents the main test configuration screen, the **Plugin Config** tab. The application is built on a plugin system, therefore each plugin in Figure 2 represents one type of attack. An attack can include one or more malicious requests that are sequentially executed at runtime (future versions of the tool will allow to select between sequential, random or user specific orders of execution for each request composing an attack).

WSFAggressor allows individual or batch selection of plugins in the same manner as WS-Attacker. Each selected plugin presents information regarding the author, version, and specific attack implemented. One of WSFAggressor's new features is the presence of test control options that allow fine-tuning the security tests to better fit the user's needs. These configurable options include the time interval between requests, number of requests to be sent during the execution of a test or the maximum duration of each executed attack. These options are transversal to all plugins, however some of WSFAggressor's plugins also allow the configuration of options that are specific to a given attack. In the following paragraphs we detail all 9 attacks (defined based on previous security research studies, existing testing tools, and field experience [6], [8], [13], [14]) currently implemented in WSFAggressor, including their configuration possibilities.

In the **Coercive Parsing** attack the SOAP body of the request includes a large quantity of deeply nested XML tags named after the operation arguments names. The default nesting depth is 100000 levels. However, the user can set this value according to its preferences. The **Malformed XML** attack consists of a set of XML malformations that are included in each malicious request (e.g., tags open but not closed, invalid characters). This attack (based on one found in SoapUI) offers no specific configuration to the user. The **Malicious Attachment** attack is based on sending a large quantity of binary data within a SOAP request. By default, WSFAggressor includes a binary zipped file attachment (a total of 100MB); however, the user can specify another file with dimensions that are more adequate to the testing goals.

The **Oversized XML** attack is based on the inclusion of very large XML elements that are sent in a malicious SOAP request. This malicious request can include 3 types of oversized XML elements: 1) Large XML tag names; 2) Large values enclosed in regular tags; 3) Large attribute names. This attack is currently stored in three files that store each of the three attack variants, occupying approximately 1.9MB each. The malicious content can be configurable by simply changing these attack signature files.

The **XML Document Size** is conceptually similar to the previous one. A very large and valid XML (SOAP) content is sent in the SOAP header or body. In similar way to the O*versized XML* attack, the malicious signature is stored in a local file with an approximate size of 1.9 MB. The user can easily change this size with a simple text-editing tool. In the **Soap Array Attack** a large array is sent in a malicious request. In some cases the web service may reserve space for that large amount of array elements in memory, which can lead to memory exhaustion of the attacked system. The user can define the default size for the array.

The **Repetitive Entity Expansion** attack recursively defines DTD entities, which the XML parser expands into a set of large entities. For instance, the expansion of 100 references to a 3-byte entity, with each reference defined in terms of the previous one, can result in a total of $(2^{101}-1) * 3$ bytes, i.e., occupying 7e+21 GB in memory. This attack is stored in a signature file and the user can change the amount of references (or size of the defined entity). The **XML Bomb** attack consists of a combination of 3 types of requests that include: 1) A definition of a large external DTD entity (e.g., 100Mb) that is loaded by the framework; 2) A definition of a large entity (hundreds of Kb) which is referenced thou-

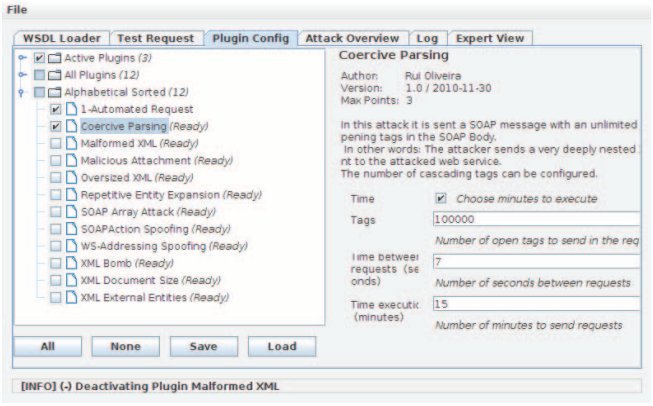


Figure 2. Plugin selection and configuration

sands times in sequence in the request; 3) A compact recursive definition of DTD entities, which the XML parser expands to a large set of entities. This attack was extracted from soapUI and currently offers no specific configuration.
Finally, WSFAggressor implements the **Xml External Entities** attack. This attack consists of a set of requests that reference well-known system files (e.g., */etc/password*) and targets information disclosure or allocation of server resources. The user can add more files to this attack by changing the respective attack signature file in the application's *customSecurityScans/attacks* folder. WSFAggressor also supports regular (i.e., non-malicious) web service invocations through the **Automated Request** plugin. This plugin can thus be used to understand the behavior of the service framework when in presence of regular requests.

After selecting the attacks that WSFAggressor will perform during the execution of a security test, the user can use the **Attack Overview** tab (see Figure 3) to review their execution order before starting the test. The user can start a security test in this tab. The security test ends when all plugins complete their execution, when the test exceeds the maximum time or maximum request count specified by the user in the test configuration options, or when the user explicitly aborts it. A new feature in our tool is the storage of test information (request identification, response content, response time, etc.) in CSV files, quite useful for large experiments.

2.2 Comparing WSFAggressor with other Tools

In this section we present a comparison between WSFAggressor and its major competitors: SoapUI [13] and WSFuzzer[14]. Note that we excluded WS-Attacker from the comparison, as it does not implement any of the attacks provided by the three tools and the attacks it includes do not exploit malicious payloads. Also, security tools focusing on application-level vulnerabilities (e.g., SQL Injection, XSS) are out of the scope of this comparison.

We selected a set of criteria with the goal of positioning WSFAggressor among competitors. The first criterion refers to the type of attacks that are supported by each of the tools. The second criterion refers to key properties that should be considered when selecting a black-box security testing tool [15]. These properties are: **1) Customization**: Characterizes a tool in terms of its configuration possibilities. In particular, how the tool handles the selection of the available attacks, how they can be combined, or how they can be configured (e.g., to execute for a given amount of time, until a certain amount of invocations); **2) Ease of use:** The tool should be intuitive and easy to use, even for users with scarce experience in security testing. Tasks should be accomplished quickly, assuming basic user competences; **3) Extensibility:** The tool should allow native add-ins or extensions that connect to third party applications. Such extensions must be easy to maintain even when the application is updated; **4) Vendor Support:** The tool provider releases upgrades on a regular basis, providing software patches for bug correction or frequent updates for feature extension.

We defined a numeric scale to describe how well the tools fulfill the above properties. The scale is based on a ranking system from 1 to 3 (a higher value is better) we chose this scale granularity as it is large (and therefore easy to apply) but still fine enough to distinguish the tools. Table 1 summarizes the comparative study targeting soapUI, WSFAggressor, and WSFuzzer and, as we can see, WSFAggressor supports more attacks than the union of the remaining tools and generally scores higher in the tool selection properties. Note that, we opted to not rank WSFAggressor's Vendor Support property in Table 1 since this is the first release, although we plan to continue actively developing it.

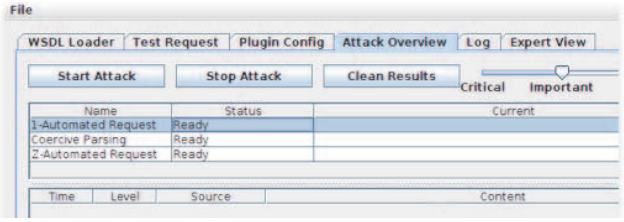


Figure 3. Attack overview and execution.

3. WSFAGGRESSOR ARCHITECTURE

This section introduces the WSFAggressor architecture. From a conceptual point-of-view it is composed of three layers: Core Layer, Plugins Layer, and SOAP Engine layer. Each layer is consists of a set of components with well-defined functions (described in the following paragraphs). Figure 4 represents a view of this layered organization.

The **Core Layer** represents the core functionality that WSFAggressor inherits from WS-Attacker. This layer is based on a Model-View-Controller (MVC) software architectural pattern [16]. The Controller (the GUIController component in Figure 4) interacts with two components on behalf of the client: A Model (TestSuite in Figure 4): which is the component responsible for holding application data; and a View (GUIView) which renders the model in the graphical interface. In this case, the Model (TestSuite component) stores web service information from the WSDL file (operation, request and interface). GUIController also calls an additional module responsible for managing and loading the default plugin structure referred as PluginManager. It invokes all plugins that implement the AbstractPlugin component. This component invokes TestSuite, retrieves the web service stored data, and delivers it to the implemented plugins, which extend AbstractPlugin.

The **Plugins Layer** extends WS-Attacker with a set of plugins that represent the core functionality of WSFAggressor. These (represented by the AttackPlugins module in Figure 4), correspond to the implementation of the attacks supported by WSFAggressor. In essence, two mechanisms are used. WSFAggressor can use automatic generation of attack signatures (user configured) or explicit load of external signatures at runtime. The attacks that have low memory requirements use dynamic generation of attack signatures based on the user configuration (specified in the graphical interface). There are, however, attacks that are static as their generation on demand would require a large amount of memory. These attacks are stored externally in signatures (represented as WSFAggressor signatures in Figure 4) and loaded at runtime. All signatures supplied with the application are stored in text files and can be reused or extended to create other plugins.

Table 1. Tool comparison.

	Criteria	Tools		
		soapUI	WSFAggressor	WSFuzzer
Supported Attacks	Coercive Parsing		X	X
	Malformed XML	X	X	
	Malicious Attachment	X	X	
	Oversized XML		X	X
	Soap Array Attack		X	
	XML Bomb	X	X	
	XML Document Size		X	X
	Repetitive Ent. Expansion		X	
	Xml External Entities		X	X
Properties	Customization	3	3	1
	Ease of Use	2	3	1
	Extensibility	2	3	2
	Vendor Support	3	-	2

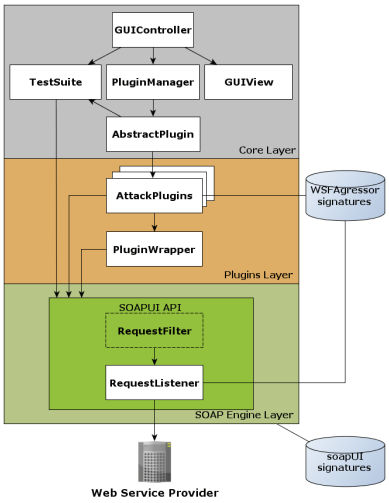


Figure 4. Internal architecture of WSFAggressor.

PluginWrapper is the other component included in the Plugins layer. Its goal is to enable the execution of frontend plugins – components with minimal internal logic that can be used to execute attacks already developed by third party developers. For instance, the XMLBomb attack is a frontend plugin to the attack with the same name already implemented by the SOAP Engine Layer.

Finally, the **SOAP Engine Layer** uses the soapUI application engine via its API. SoapUI's libraries are invoked on behalf of WSFAggressor to send and receive SOAP messages to a web service Provider. In short, TestSuite invokes soapUI to retrieve and store the web service interface. AttackPlugins invoke soapUI to send the malicious requests (and also to receive the responses) to the web service provider. The PluginWrapper interaction with SoapUI enables the execution of soapUI's internal security attacks. Since SoapUI only supports security attacks since version 4.0, WSFAggressor, features an updated soapUI engine (version 4.0.1) compared to the engine that what was originally provided in WS-Attacker (version 2.5).

The SOAP Engine layer contains (in addition to internal soapUI components) the RequestListener component (specifically built for WSFAggressor). This module implements the RequestFilter interface from the soapUI API and acts as a listener to all outgoing SOAP requests. When WSFAggressor executes an attack, the request containing the malicious content is passed to the lower layers until it reaches the soapUI libraries. However, the structures of some attacks is not expected by soapUI and generates internal errors (aborting the dispatch of the request). In these cases the application includes a unique token in the request (corresponding to the attack). RequestListener is then called immediately before the request is sent to the provider (at the HTTP transport layer), and places the corresponding WSFAggressor attack signatures (replacing the token) in the HTTP SOAP payload.

To create a new attack, a developer simply needs to create a class that extends *AbstractPlugin* and place it in the *wsfaggressor.plugin* package. The methods required to implement are straightforward and are described in detail in [11]. The main task to perform is to implement the *attackImplementationHook(RequestResponsePair)* method, which allows the developer to extract a regular (non-malicious) request object from the method argument and manipulate it as desired, with the help of easy-to-use methods.

4. WSFAGGRESSOR DEMO

In this section we present a set of preliminary experiments, designed to illustrate the security testing capabilities of WSFAggressor. It is important to emphasize that the experiments are a subset of a larger evaluation described in [17], where WSFAggressor was used with great success. In the next sections we present the experimental setup and the services selected for the experiments. We then describe WSFAggressor's default execution profile (used in these experiments) and conclude the section by presenting and discussing the main results obtained.

4.1 Experimental Setup and Tests Monitoring

To illustrate WSFAggressor's capabilities we deployed the tool in a client machine and executed it against a test service (see details in the next section) deployed in a server machine configured with **Tomcat 7.0.23** [18] two frameworks: **Apache Axis 1** (version 1.4.1) [19] and **Apache Axis 2** (version 1.6.1) [3]. We selected Tomcat due to its high use and popularity among developers and providers, Axis 1 due to the fact that it is a highly matured service framework, still currently used in many production systems, and Axis 2 due to its popularity. The 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 infrastructure software.

During the experiments, the Tomcat server was continuously monitored using JConsole 1.6.0_30-b12 [20]. Based on previous studies [7], [21], we observed the following parameters: JVM memory heap size, number of allocated threads, and CPU usage. All experiments where an abnormal behavior was observed (e.g., high memory usage) were repeated up to a total of 3 executions, with the goal of verifying possible deviations. No significant deviations were detected during the experiments presented in this section (full details regarding all tests as well as JVM configuration flags can be found at [11]).

4.2 Test Service Design and Invocation

In order to attack a framework, we need a service on top of it so that we can exercise the underlying framework. We implemented a test service that includes four operations with common input types (Integers, Strings, and Arrays and File attachments). As our target is not the web service implementation there are no representativeness requirements involved with the actual code for the service described above (we simply need an entry point to the framework, and this is provided by the web service interface).

WSFAggressor also supports sending regular (i.e., non-malicious) service invocations. The values used in these invocations were chosen based on the maximum values found in all operations defined by the WSTest and TPC-App benchmarks [22], [23]. These values (used with the getInt, getString, getArray, and getFile operations, respectively) are: 10, a 6-byte random String, two hundred 6-byte random String array elements, a 700Kb JPEG file.

4.3 WSFAggressor execution profile

WSFAggressor includes a default procedure that can be used to assess a given service framework in terms of security. This pre-

Table 2. Infrastructure supporting the experiments.

Node	Software		Hardware	
	Operating	***Java VM***	***CPU***	***RAM***
Client	Ubuntu Linux 10.04 (32 bits)	OpenJDK 1.6.0_20	Intel core 2 Duo T6500 (2.1GHz)	3 GB
Server	Windows XP (64 Bits)	Oracle Java 1.6.0_30-b12	AMD Athlon X2 Dual Core 4200 (2.21GHz)	4 GB

Table 3. Attacks and Observed Problems.

Attack Type	Failures	Abnormal behaviors
Coercive Parsing	Axis1	
OversizedXML		Axis1
SoapArrayAttack	Axis1	Axis2
XMLDocumentSize		Axis1

configured procedure includes sending malicious and non-malicious (i.e., regular) requests to the server in three distinct phases: Pre-attack; Attack; and Post-Attack. This is illustrated in the timeline represented in Figure 5 and described next.

As we can see in Figure 5, the **Pre-attack phase** includes two periods, a warm-up period where no requests are sent to the service and a regular invocation period where non-malicious requests are sent to the web service. These two periods are preset to 5 minutes each; however, the application user can define any given amount of time that is adequate for the system being tested. The goal of the Pre-attack phase is to understand the behavior of the web service platform when idle and when handling normal requests, so that we can compare that behavior with the one observed during or after attacking the server. After this first phase, the **Attack phase** follows and one or more attacks are sent to the service during a configurable amount of time (set to 15 minutes by default). This phase serves the purpose of understanding how the service framework behaves when directly in presence of a security attack. Finally, the tool executes (in this pre-configured environment) a **Post-attack phase**, which includes two periods: a regular invocation period, where non-malicious valid requests are sent to the server and a rest period where no requests are sent to the provider. The goal of this phase is to detect if the attack phase has any effect on the regular service platform operation. The regular and rest periods are preset with a duration of 5 and 60 minutes. Regard that this pre-configured tool profile can be completely changed according to the users goals and by performing a small set of configurations in the tool. The pre-configured durations were empirically defined, are adequate to understand the existence of issues (as shown in the next section), and also should be kept practical, since using high durations is usually not an option for developers that frequently have time limits for test phases.

Each experiment performed (which had a total duration of 95 minutes) used a single attack type, and the system was restored between experiments to avoid the propagation of tests effects. The attacks performed by WSFAggressor were configured with the default values defined in Section 2.1. Each client request (malicious or not) was sent in a synchronous and non-parallel fashion every 7 seconds after the reception of each response [24]. Each request's timeout (i.e., the time the tool waits for a service response) was set to 1 hour to detect the cases where web service does not provide a timely response.

4.4 Results and Discussion

In this section we discuss the main results obtained using WSFAggressor against well-known web services stacks deployed in a popular container (i.e., Axis 1, Axis 2 and Tomcat). As referred, the 9 currently available attacks types in our tool were executed in distinct experiments that were repeated up to a total of 3 executions. Table 3 summarizes the observed problems and maps them to their originating attack. As shown, 4 out of the 9 attacks executed were able to trigger a total of two failures and three abnormal behaviors in the platforms. In the remaining, no significant deviation from the normal behavior was observed. The following paragraphs describe the observed issues.

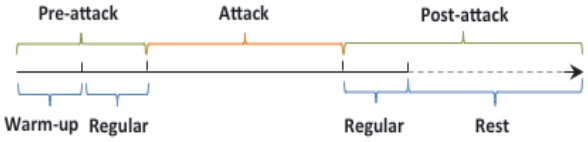


Figure 5. Approach for assessing frameworks security.

An **abnormal behavior** was detected when executing the tests against the Axis 2 framework. The *Soap Array Attack* leads this framework to consume around 400Mb of memory in the attack period (see Figure 6). Despite the observed behavior for Axis 2, the framework was able to recover to normal values around 1 hour after starting the rest period. Axis 1 shows greater difficulties when handling this attack, essentially doubling the amount of memory required, using more time to reply to the client and turning unresponsive during the attack period.

A **failure** was observed when executing the *Soap Array Attack* (see Figure 7). Short after sending the first attack, the allocated memory rises and maintains itself close to 750Mb, with the CPU achieving an approximate average of 50% usage, with sporadic peaks reaching 100%. In the meantime, WSFAggressor remains idle, waiting for a response and the Tomcat logs report the occurrence of *OutOfMemory* exceptions, with indication of failure of internal Tomcat components. After about 8 minutes an *OutOfMemory* exception is delivered to the client, which then issues another attack. The same behavior is observed and is followed by a steep decrease of CPU and memory usage (short after the start of the Post-Attack phase). We tried to check if, during these abnormal periods, the server was still responsive and we issued a regular request to the Axis service with an external tool, with no response being obtained. We also tried to check if this failure affected other services in Tomcat, and issued a new request to another framework (which was deployed in Tomcat for this verification purpose only), with no response being obtained. Ideally, the container should provide some isolation among applications, which was not the case. Finally, we performed an extra verification and tried to see if Tomcat's web page service engine could still serve HTTP requests (we issued a request to Tomcat's default webpage), which was not also the case. Despite the unresponsiveness period and failure occurrence, we observed that the platform was able to recover after handling each attack during 8 minutes.

Another **failure** was observed when executing the *Coercive Parsing Attack* targeting the *getInt* operation. As shown in Figure 8, the CPU usage reaches high values, reaching around 50% during the attacks and 100% in the 10 minutes that immediately follow the attack phase (this also occurs with the *getString* operation). During these 10 minutes there is a continuous output of a *StackOverflowException* to the server logs, which is an indicator of the occurrence of an internal error, but is clearly classified as **failure**.

Finally, when executing the experiments with the *Oversized XML* and *XML Document Size* attacks we detected two **abnormal behaviors** associated with high memory allocation for a prolonged period of time. As we can see in Figure 10 and Figure 11 (that

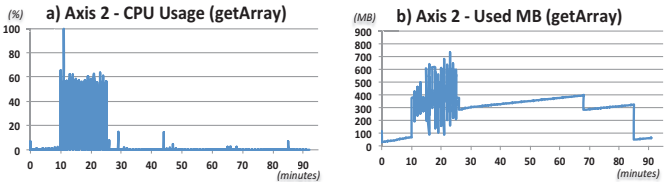
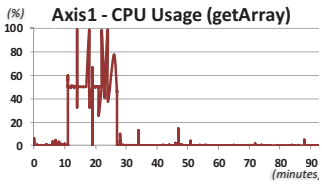
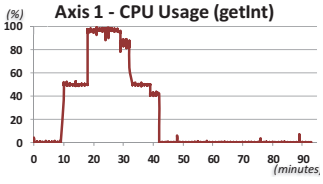
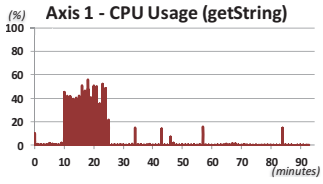
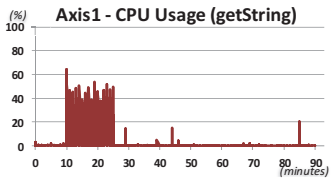


Figure 6. Axis 2: Soap Array Attack.









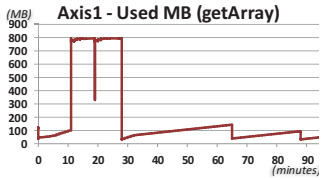


Figure 7. Soap Array Attack

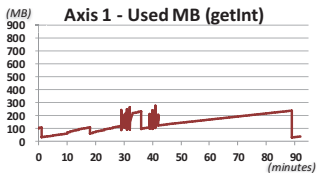


Figure 8. Coercive Parse Attack

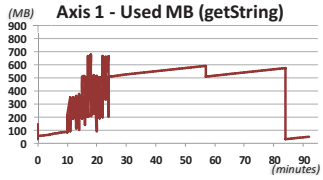


Figure 9. Oversized XML Attack

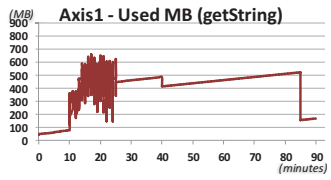


Figure 10. XML Document Size Attack

represent *Oversized XML* and *XML Document Size* experiments, respectively) the CPU usage increases during the attack period, returning to regular values after the Attack phase (with small sporadic usage peaks). The most critical aspect is the memory allocation pattern, which in both Attack phases surpass the 500 MB mark. Moreover, these high values of allocated memory subsist in the Post-Attack phase, with the memory only being released by the end of the observation period. Obviously, this is far from being an ideal behavior since it diminishes the resources that could be available for use by the framework or other applications deployed in the server. Also note that the test environment is based in non-parallel requests. Multiple simultaneous requests may result in a higher impact in the platform.

5. CONCLUSION

This paper presented a tool for web service frameworks security testing. WSFAggressor fills a gap in current security tools, providing a large set of attacks specifically selected to target service frameworks. The tool is freely available and requires little configuration effort and expertise knowledge to be used. WSFAggressor currently supports 9 attacks, which is (to our best knowledge) more than the sum of the attacks supported by current security testing tools (in the same category). We illustrated the tool's capabilities using popular service platforms. The 2 failures and 3 abnormal behaviors observed indicate that corrective measures are urgent and obviously show the tool's usefulness. The tool can be quite useful for providers to assess the security of their service platforms, but also for developers to disclose potentially severe issues before deployment.

6. REFERENCES

[1] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, "Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI", *IEEE Internet Computing*, vol. 6, n. 2, pp. 86-93, 2002.

[2] G. U & S. Rao, "Develop Web services with Axis2: Deploy and consume simple Web services using the Axis2 runtime". ibm.com/developerworks/opensource/library/ws-webaxis1/.

[3] "Apache Axis2". https://axis.apache.org/axis2/java/core/.

[4] "Metro". [Online]. Available: http://metro.java.net/.

[5] "Apache CXF". https://cxf.apache.org/.

[6] M. Jensen, N. Gruschka, and R. Herkenhöner, "A survey of attacks on web services", *Comp. Sci. Res. Dev.*, May 2009.

[7] S. Suriadi, A. Clark, and D. Schmidt, "Validating Denial of Service Vulnerabilities in Web Services", in *International Conference on Network and System Security (NSS), 2010*.

[8] Intel SOA Expressway, "Protecting Enterprise, SaaS & Cloud-based Applications - A Comprehensive Threat Model for REST, SOA and Web 2.0".

[9] N. Antunes, N. Laranjeiro, M. Vieira, and H. Madeira, "Effective Detection of SQL/XPath Injection Vulnerabilities in Web Services", *Intl. Conf. on Services Computing,SCC 2009*.

[10] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell, "State of the Art: Automated Black-Box Web Application Vulnerability Testing", *Symposium on Security and Privacy (SP)* 2010.

[11] R. Oliveira, N. Laranjeiro, and M. Vieira, "WSFAggressor and results". http://student.dei.uc.pt/~racoliv/papers/mid.zip.

[12] "WS-Attacker", http://sourceforge.net/projects/ws-attacker/.

[13] "soapUI - Functional Testing". http://www.soapui.org/.

[14] "WSFuzzer". owasp.org/index.php/Category:OWASP_WSFuzzer_Project.

[15] C. C. Michael and W. Radosevich, "Black Box Security Testing Tools". https://buildsecurityin.us-cert.gov/bsi/articles/tools/black-box/261-BSI.html.

[16] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[17] Rui A. Oliveira, N. Laranjeiro, and M. Vieira, "Experimental Evaluation of Web Service Frameworks in the Presence of Security Attacks", Intl. Conf. on Services Computing 2012.

[18] "Apache Tomcat". https://tomcat.apache.org/.

[19] "Apache Axis". https://axis.apache.org/axis/.

[20] "Using JConsole to Monitor Applications". java.sun.com/developer/technicalArticles/J2SE/jconsole.html.

[21] Gang Wang, Cheng Xu, Ying Li, and Ying Chen, "Analyzing XML Parser Memory Characteristics: Experiments towards Improving Web Services Performance", in *International Conference on Web Services (ICWS), 2006.*

[22] "Comparing Java 2 EE and .NET Frameworks", 2004. http://java.sun.com/performance/reference/whitepapers/WS_Test-1_0.pdf.

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

[24] S. Ranjan, R. Swaminathan, M. Uysal, A. Nucci, and E. Knightly, "DDoS-shield: DDoS-resilient scheduling to counter application layer attacks", *IEEE/ACM Trans. Netw.*, vol. 17, n. 1, pp. 26–39, Feb 2009.