A new Approach towards DoS Penetration Testing on Web Services

Andreas Falkenberg
SEC Consult Deutschland
Unternehmensberatung GmbH, Germany
a.falkenberg@sec-consult.com

Christian Mainka, Juraj Somorovsky, Jörg Schwenk,
Horst Görtz Institute for IT Security
Ruhr University Bochum, Germany
{Christian.Mainka, Juraj.Somorovsky, Joerg.Schwenk}@rub.de

***Abstract*—SOAP-based Web services is a middleware technology marketed as the solution to easy data exchange between heterogeneous IT architectures. The large number of scenarios, in which this technology is used, has introduced demands for new extensions raising its complexity. However, this has also introduced a large variety of new attacks.**

In this paper, we investigate an automatic evaluation of Web service specific Denial of Service (DoS) attacks. We present a new fully automated plugin for the WS-Attacker penetration testing tool implementing major DoS attacks. Our tool determines the attack success without having physical access to the target machine, using a novel blackbox approach. We give an overview of our design decisions and present the evaluation results using common Web service frameworks and systems.

***Index Terms*—SOAP-based Web services, WS-Security, Denial-of-Service, Penetration Testing Tool, WS-Attacker**

I. Introduction

Web services are marketed as the solution to all interoperability issues between heterogeneous systems. The Web service paradigm is a concept composed of an array of new standards based on established Internet technologies and protocols, such as XML [1], SOAP [2], and HTTP. The flexibility demands in this paradigm caused continuous development of new standards addressing e.g. secure data exchange [3], identity management [4], or policies [5].

Due to a wide array of complex Web services standards, a new class of attacks emerged over the last years. These Web service specific attacks strictly target the Web service processing stack. It has been shown e.g. that it is possible to break integrity, authenticity [6], [7] and confidentiality of the exchanged messages [8], [9], or availability of the Web services systems [10].

Attacks aiming at the availability of a computer resource – so called Denial of Service (DoS) attacks – gained lot of attention in the recent time. With many recent high profile DoS attacks on companies such as VISA [11], PayPal [12] and various government agencies [13], DoS attacks pose a serious threat to today's IT infrastructure. Depending on the impact of a DoS attack, the response time of the target is slowed down – or even worse – the availability of the computer resource might be interrupted temporarily. For example, very recent DoS attacks based on weak hash-mapping algorithm implementations (called *HashDoS* attacks) showed that a huge impact can be achieved with only small request messages [14], [15], [16].

DoS attacks can be performed using a huge number of different techniques. Especially those targeting systems using XML-based message formats can misuse different XML properties. Due to the complexity of XML documents and the resource intensive nature of XML parsing, even a small malformed message can be used to occupy a significant amount of resources. The DoS attacks on Web services can exploit parsing mechanisms, transformations, or resolution of external entities[1]. Table I lists common Web service DoS attacks.

Coercive Parsing	SOAP Array Attack
Oversized XML Attack	Hash Collision Attack
XML External Entity DoS	XML Entity Expansion
Oversized Cryptography	Recursive Cryptography
XML Signature: Transformation DoS	XML Signature: Key Retrieval DoS

TABLE I: Web service specific DoS attacks.

With deployments in critical infrastructures [17], [18] and major industries [19], Web services are an attractive target for attackers aiming at the availability. Thus it is of crucial importance for Web service developers and providers to possess an automated penetration testing tool that automatically evaluates whether a Web service is vulnerable to these attacks or not. This motivated us for the development of a new DoS attack plugin for the WS-Attacker framework [20][2].

Contribution. In this paper, we first present general requirements for the evaluation of DoS attacks. Thereby, we assume that the attack executor (pentester) does not have any possibility to run a specific program on the attacked system. He can only send his payloads to the server and evaluate the response times. Based on this assumption, we developed an approach supporting such evaluations. Basically, our plugin works in *two phases*: In the first phase, our plugin sends to the server *untampered* payloads. In the second phase, *tampered* requests with attack payloads are sent. In a *parallel* thread, the plugin simulates a 3rd party client that continuously sends request to the tested Web service during the first and the second phase. The timings of all requests are logged. Results of these measurements are automatically evaluated to compute and depict the resulting attack impact.

[1] http://ws-attacks.org
[2] http://sourceforge.net/projects/ws-attacker/

We implemented the above described approach and several Web service specific DoS attacks as a WS-Attacker plugin. We evaluated our plugin using five major Web service frameworks: Apache Axis2 Java [21], Apache CXF [22], IBM Datapower [23], Metro [24], and .NET [25]. Our results show that the DoS penetration testing tool works as expected, giving a clear indication whether or not a tested Web service is vulnerable – without false positives. For example, our tests revealed that the latest versions of the Axis2 Java, Apache CXF, and the Metro Framework were vulnerable to the implemented attacks.

Related Work. Nowadays, there exists a large number of penetration testing tools for Web applications. However, most of the tools concentrate on the typical attack threats such as XSS, SQLi, or typical DoS attacks. Web service specific DoS attacks are not in scope of those tools.

The closest relation to our approach can be found in the tools SoapUI[3], WSFuzzer[4], and WSFAggressor [26]. SoapUI and WSFuzzer are Web service frameworks developed specifically for testing Web services platforms. However, these tools support only few DoS specific attacks and do not introduce any specific attack evaluation.

Very recently, an approach for automated penetration evaluation of Web service specific attacks has been published by Oliveira et al. [26]. As a result of their research, they implement a tool called WSFAggressor, which is also based on WS-Attacker, containing a large number of DoS attacks. However, in order to evaluate the attack success, this tool requires access to the tested system. This prerequisite is not given by evaluating specific hardware devices such as IBM Datapower [23], or pentesting sensitive customers' servers. Moreover, this tool misses some important attack techniques such as HashDoS [14]. In contrary, our tool implements a new approach that allows a security expert to easily and accurately decide whether a Web service is vulnerable to the major Web service specific DoS attacks – without having an access to the tested system.

II. Foundations

In the following, we present the relevant underlying technologies of Web services.

A. Extensible Markup Language (XML)

XML [1] is a self-describing, universal, and platform-independent markup language. It is the most common format for the transmission of data between heterogeneous applications. Listing 1 shows an example XML file, which stores a fictitious order in a bookshop.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<order>
    <customerID>1234</customerID>
    <ISBN>1234567234-3</ISBN>
    <Price>10.00    </Price>
</order>
```

Listing 1: An exemplary XML file.

[3]http://www.soapui.org

[4]http://sourceforge.net/projects/wsfuzzer

An XML file can be prepended by a Document Type Definition (DTD) [1], which can be used to define the structure of the XML document, or to declare custom internal or external entities.

B. XML Parsing

There are various approaches to parse an XML document. DOM-based parsers create a tree structure of the whole XML document, called the document object model (DOM). Any programmer accessing the XML document can easily traverse through the XML tree, access, insert, and delete nodes. A node can be an element, an attribute, a text content, or a comment. The main disadvantage of this parsing approach is the high resource utilization. The size needed for an XML tree representation exceeds the original size multiple times. The whole document must be read before the programmer can access its contents.

SAX parsers are event-based. A SAX parser only operates on parts of the XML tree at a time. Whenever the parser encounters an XML node, an event is triggered. It is the program's task to handle these events in order to process the XML document. The main advantage of event-based parsing is the low utilization of resources. Furthermore, once an event is triggered the event handlers can immediately start processing the data, without having to wait for the parser to reach the end of the XML document. However, some operations will get more complex compared to a DOM parser, e.g. sorting an XML document requires multiple passes since the SAX parser only reads sequentially forward.

StAX parsers also use the streaming based approach. However, instead of automatically triggering events while parsing the document, the StAX parser waits for method calls that trigger certain parsing operations.

C. SOAP Based Web Services

SOAP is an XML standard used to exchange messages between Web services. It is completely independent from the underlying transport protocol. Predominantly, the HTTP-Protocol is used for transporting SOAP messages. A SOAP message is composed of an `Envelope` element that contains a `Header` and a `Body` element. The SOAP header contains directives for achieving security objectives such as confidentiality and integrity. The SOAP body contains the action to be performed. Listing 2 shows a simple SOAP message, which queries a stock quote service.

```
<?xml version="1.0"?>
<soap:Envelope
  <soap:Header/>
  <soap:Body>
    <stockQuote>
      <symbol>GOOG</symbol>
    </stockQuote>
  <soap:Body>
</soap:Envelope>
```

Listing 2: An exemplary SOAP message.

WSDL files are used to describe the Web service and the structure for SOAP message exchange. The interaction between server and client is shown in Figure 1. The process

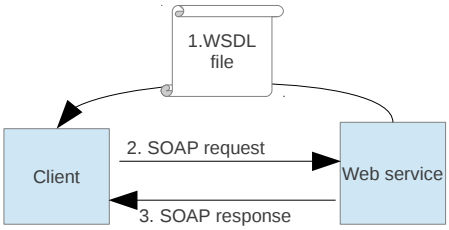

Fig. 1: Web service message exchange pattern.

is started by receiving a WSDL file. Based on the WSDL file a SOAP message is generated by the client and sent to the Web service. The Web service processes the request and then returns a SOAP response.

D. Processing a Web Service Request on Server Side

In order to define the attack surface for Web service specific attacks the life cycle of a Web service request is shown. Figure 2 explains the inner workings of the receiving Web service in detail. The workflow is based on the Web service framework Apache Axis2 [21].

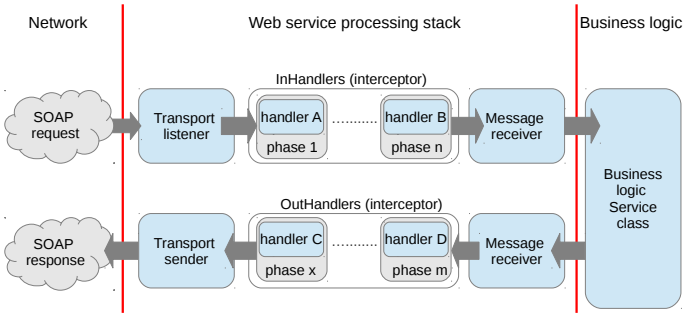

Fig. 2: Web service message flow based on the Axis2 framework.

A SOAP request coming from the network is first processed by the transport listener. The SOAP request is then passed through a pipeline of predefined or custom handlers. Each handler is responsible for a specific message processing part. For example a security handler decrypts encrypted message parts and validates signatures. If the SOAP message was passed through all the handlers successfully, the message receiver receives the SOAP message and hands the relevant data over to the application logic.

The exact names of the process steps and their order heavily depend on the used Web service framework. Even though the processing might differ depending on the framework, the main concept doesn't change. Web service specific attacks are targeting the Web service processing pipeline. Attacks affecting a component outside the Web service processing stack are not considered.

III. Web Service specific DoS Attacks

Denial of Service (DoS) attacks come in a wide array of variations. The entire application stack of a system can be vulnerable to DoS attacks at some point. However, as described in Section II-D, only DoS attacks are considered that attack the Web service processing pipeline. DoS attacks such as SYN-Flooding or DoS attacks on the database system behind a Web service are not Web service specific and therefore not considered.

In the following, a brief description of common Web service specific DoS attacks is given:

1) Coercive Parsing Attack
   The attacker sends an XML document that contains a deeply nested XML structure. When a DOM-based parser processes the XML document, an out of memory exception or a high CPU load can occur [27].
2) SOAP Array Attack
   The SOAP Array Attack forces the attacked Web service to declare a very large SOAP array. This can exhaust the memory of the attacked Web service [28].
3) XML Attribute Count Attack
   The XML Element Count Attack attacks the server by sending a SOAP message with a high attribute count [29].
4) XML Element Count Attack
   The XML Element Count Attack attacks the server by creating a SOAP message with a huge number of non-nested elements [27].
5) XML Entity Expansion Attack
   The XML Entity Expansion Attack causes a DoS scenario by forcing the server to recursively resolve entities defined within a Document Type Definition (DTD). This attack is also known as XML Bomb or Billion laughs Attack [30].
6) XML External Entity DoS Attack
   The XML External Entity Attack causes a DoS scenario by forcing the server to resolve a large external entity defined within a DTD. Please note that if an attacker is able to execute the External Entity attack, an additional attack surface might appear [31].[5]
7) XML Overlong Names Attack
   In an XML Overlong Name Attack an attacker injects overlong XML nodes in the XML document. Overlong nodes can be overlong element names, attribute names, attribute values, or namespace definitions [29].
8) Hash Collision Attack – HashDoS
   A hash table is a data structure, which maps arbitrary keys to values. Within XML documents hash tables are e.g. used to store attributes and their corresponding values or XML namespace definitions of an element. The basic hash table principle is shown in Figure 3. The value of a certain key is mapped to a storage bucket through a hash function that takes the key as input. Ideally each key should result in a unique bucket. However in some cases different keys result in the same bucket assignments, causing a collision. A collision will lead to resource intensive computations within the bucket. When a weak hash function is used an attacker can intentionally create hash collisions that will lead to

[5]http://www.agarri.fr/blog

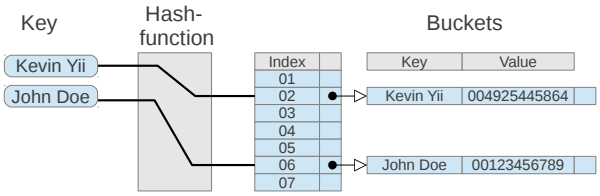

Fig. 3: Hash table principle.

a DoS scenario [14].

A good evaluation of the attacks' impacts on the memory and CPU usage on the Web services servers is given in [26].

IV. Automatic testing of DoS attacks for Web Services

In this section, we present general requirements and the resulting concept behind the newly created Web service specific DoS penetration testing tool. We explain the internal program workflow during the attack execution, its implementation, and the GUI presenting the attack results.

A. Requirements

Various requirements have to be met by a Web service specific DoS penetration testing tool to generate reliable results.

Measuring Attack Success Using a Blackbox Approach: Since the tester has no physical or virtual access to the targeted system, the attack success can only be decided by observing the server response time RT. The RT is defined as follows:

$$T_{\text{RT}} = \left| T_{\text{FirstByte}_{\text{Server}}} - T_{\text{LastByte}_{\text{Client}}} \right|$$

Measurement of the RT begins as soon as the last byte of the request is sent to the server. It ends when the first byte of the response from the server is received. We refer to this scenario as the blackbox approach. The blackbox approach is chosen because it minimizes the prerequisites to run the tool and maximizes the field of application.

Automated Crafting and Sending of Attack Messages: The automated penetration test starts with the tester picking its target and choosing the DoS attack he wants to run against the target. If required for the attack to function properly, attack specific parameters have to be entered by the tester. From there on the entire vulnerability testing process has to be performed automatically without any user interaction required. This includes the automated crafting and sending of messages with attack payloads based on the parameters supplied by the tester. These requests are referred to as tampered requests in the following. On the other hand, regular requests are referred to as untampered requests.

Fitness for Various Load Patterns: Depending on the goal of the tester, an attack will be run under various load patterns. In order to allow this flexibility, fitness for various load patterns is required. During initialization of the attack, the tester should be able to set:

- Number of threads sending tampered requests in parallel.
- Number of tampered requests per thread.
- Milliseconds between sending each tampered request.
- Attack specific parameters – only if applicable – e.g. length of payload.

Please note that our tool was designed to be run from a single machine. Thus, the following restrictions apply:

- All requests are sent from the same IP. The simulation of a Distributed Denial of Service (DDoS) attack is not possible.
- The number of parallel requests is limited by the bandwidth of the machine running the Web service specific DoS penetration testing tool.

However, DDoS simulation was not in scope of our work and these restrictions should not prevent a penetration tester from testing. An effective DoS attack can be mounted on a vulnerable machine with very few requests.

Fitness for Different Test Scenarios: Depending on the goal of the tester, the following two testing scenarios should be supported.

First, the tool should show the tester whether or not the tested Web service is vulnerable to the selected attack. A Web service is vulnerable to an attack if the difference between the response time RT_t of a tampered request and the response time RT_r of an untampered regular request exceeds a defined threshold.

Second, the tool should evaluate the attack effect on 3rd party users. Once a Web service was found to be vulnerable to an attack, the prerequisites for a successful DoS attack are given. However, sending a single tampered request to a vulnerable Web service is usually not enough to cause a DoS scenario that affects 3rd party users. Only if the response time of regular requests from 3rd party users increases during an attack a valid DoS attack was performed. When testing under a given load scenario, the tool should make statements in regard to the following two points:

- Are 3rd party users effected by the attack?
- For how long are 3rd party users affected (even after the attack is finished)?

Exclusion of Subattacks: Some attacks are composed of different subattacks. However a tester might want to test for only one of these subattacks. The DoS penetration testing tool has to provide mechanisms that allows this flexibility.

Elimination of Errors: Depending on the attack set up, different errors might get induced in the results of an attack. The following two main error sources are defined that get eliminated by the automated Web service specific DoS penetration testing tool.

- Errors due to server high load: Depending on the load scenario, defined in Section IV-A, the attacks might be performed with a high number of requests in parallel. Already a high number of requests might cause an increased RT or even halt the server. However, these increased RTs are not caused by the attack payload itself,

they are just caused by the shear number of requests. Therefore, an increase in RTs induced by high request numbers should be eliminated.
- Errors due to increased message sizes: When sending a tampered request, the size of the message usually increases due to the injected payload. Based on the definition of RT, an increase in message size will lead to a higher RT. This error source has to be eliminated.

Despite being able to cancel out two error sources, the following two error sources will remain:

- A significant change in network response time while attack is running due to 3rd party traffic.
- A significant change in SOAP message processing time on the server due to 3rd party traffic.

These error sources are beyond the scope of this tool. In the following, we always assure that these two errors do not occur.

B. DoS Attack Success Evaluation

Based on the above given requirements, we developed a DoS attack success evaluation. Its basic idea is presented in Figure 4. In Phase 1 a user-defined load pattern is sent

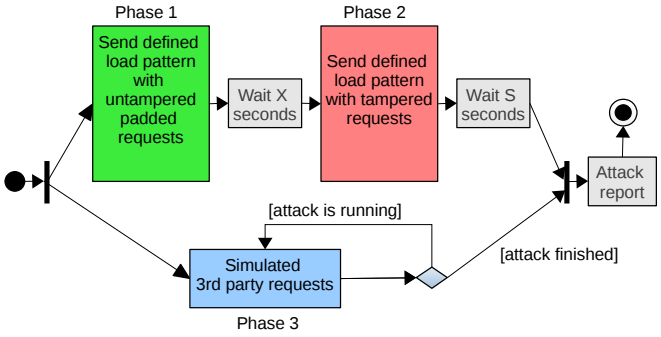

Fig. 4: Architecture of the DoS attack success evaluation used in our tool.

to the target Web service with untampered requests that are padded to the size of tampered requests[6]. After a user-defined waiting period the same load pattern is sent to the target again, but this time with tampered requests. In order to conduct a vulnerability test, the tool calculates the ratio of the median RT of tampered requests RT_t to untampered regular requests RT_r. This method will guarantee that any ratio significantly higher or lower than one must have been caused by the payload of the tampered requests. All other major error sources such as increase in RT due to different message sizes or different network loads are eliminated by the design.

In parallel to these two processes, phase 3 continuously sends test requests to the target Web service in constant user-defined intervals. These requests simulate 3rd party users that visit the targeted Web service while the attack is running. If the RT of 3rd party users increases one can speak of a successful DoS attack, not just a vulnerability test.

[6]The expansion of the message size of the untampered requests poses no problem. Even if the untampered message would cause a DoS scenario, it would show up in the attack results, effectively avoiding false positives.

To formalize the definition of attack success the following two attack success metrics are defined:

Test for vulnerability: Attack success is measured using the metric "Attack-RT-ratio" ($ARTR$). The $ARTR$ is defined as follows:

- RT_{10t}: Median of the RT of the last ten tampered requests.
- RT_{10r}: Median of the RT of the last ten untampered (regular) requests.
- Calculate the ratio:

$$ARTR = RT_{10t} \ / \ RT_{10r}$$

If less than ten (un)tampered requests are available, the maximum number of requests is taken to calculate the ratio.

An attack success is defined as follows:

Metric value:	Rating:
$ARTR < 3$	payload ineffective
$ARTR \geq 3$ and $ARTR < 6$	payload effective
$ARTR \geq 6$	payload highly effective

TABLE II: Metric "Attack-RT-ratio" $ARTR$.

The threshold values were chosen based on pretests on vulnerable and non-vulnerable Web services.

Test for Attack Effect on 3rd Party Users: Attack success is measured using the metric "3rd-Party-RT-after-attack". The "3rd-Party-RT-after-attack" is defined as follows: *Median RT of all simulated 3rd party requests after starting to send first tampered request.* Attack success is defined as follows:

Metric value:	Rating:
3rd-Party-RT-after-attack $< 2sec$	no or small effect on 3rd party users
3rd-Party-RT-after-attack $\geq 2sec$ and 3rd-Party-RT-after-attack $< 5sec$	3rd party users are affected
3rd-Party-RT-after-attack $\geq 5sec$	3rd party users are heavily affected

TABLE III: Metric "3rd-Party-RT-after-attack".

Two seconds is our threshold that is still acceptable for a response. Every RT above 2 seconds is considered a success because the users annoyance level increases exponentially.

C. Implementation

The Web service specific DoS penetration testing tool was implemented as a plugin for the WS-Attacker framework, which is an extensible open source Web service specific penetration testing framework [20]. It comes with the required functionality to process WSDL files and SOAP messages. New attacks can be added using the Java plugin architecture.

The Denial of Service plugin is split into two main components, the MVC DoS extension and the DoS attack classes. The general relation between these two components is shown in Figure 5.

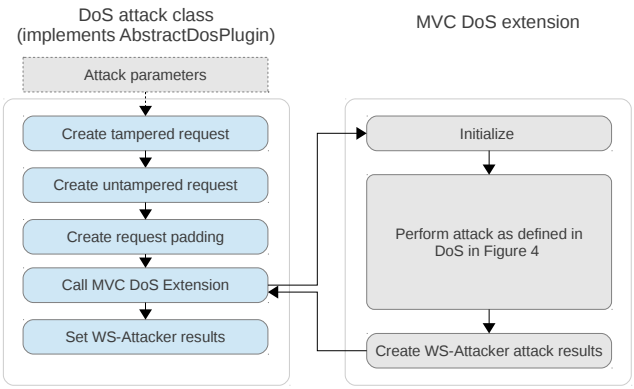

Fig. 5: Internal workflow of Denial_of_Service plugin

DoS Attack Classes: A DoS attack class provides the attack specific implementation details. This includes the following four points:

- Setup of custom attack parameters:
  Depending on the attack, custom attack specific parameters are required. They allow the tester to customize the attack payload according to its testing scenario.
- Tampered request creation:
  Every DoS attack class has to implement its own tampered request creation method by overriding the method *createTamperedRequest()*. There are no restrictions of how the request is structured. HTTP header fields can be set to arbitrary values. The SOAP request itself can be set freely. It doesn't even have to be valid XML since the request is sent via the Java class *HttpClient*[7]. The return value is an object of the class *RequestObject* that holds all data that is required for a valid HTTP request.
- Untampered request creation:
  The creation of an untampered request is not mandatory for a developer. The class *AbstractDosPlugin* already provides a default implementation. However, based on the attack type, the untampered request might have to a have a specific structure in order to cancel out certain errors. In this case, the developer has to override the method *createUntamperedRequest()* provided by the class *AbstractDosPlugin*.
- Create attack padding:
  For reasons of error correction, as defined in Section IV-A, the tampered request and untampered request have to have the same size. Hence, the smaller one out of the two requests has to be padded to the size of the larger request. The class *AbstractDosPlugin* already provides a default implementation for the request padding.

MVC DoS Extension: The MVC DoS extension provides the DoS attack-specific functionality that is required by every DoS attack plugin. It expects a plugin that implements the class *AbstractDosPlugin* as input. Once started, the DoS attack is performed according to the activity diagram defined in Figure 4. When developing a new DoS attack, there should be no need to change any part of the MVC DoS extension. All attack-dependent parts can be set in the DoS attack class that implements the attack.

[7]http://hc.apache.org/httpclient-3.x/

D. Attack Result Presentation

The results of an attack are presented to the tester in a graph and in a numerical fashion, using the two attack success metrics defined before. In the following, the results of an XML Coercive Parsing attack on a vulnerable test target are shown.

The test was conducted using slightly modified default attack parameters. During attack setup, the number of parallel threads was increased to three and the number of requests per thread was increased to ten. All other attack parameters were left at their default values (in such a configuration, the number of nested elements is set to 75,000).

The results of the attack success metric are as follows:

Metric:	Result:	Rating:
$ARTR$	1052	payload highly effective
3rd-Party-RT-after-attack	59 sec	3rd party users are heavily affected

TABLE IV: Attack Success Metric of full DoS test.

Both metrics indicate a strong attack impact.

The graphical presentation of the successful test is shown in Figure 6. The graph is automatically created by the test tool after an attack is finished. The red and the green bars indicate the number of requests sent per interval (1 second). The individual connected data points state the mean RT of all requests sent in the used interval. Figure 6 shows that RT_t (response time by tampered requests) is significantly higher than RT_r (response time by regular requests), proving that the target is vulnerable and verifying the results of $ARTR$. Furthermore, Figure 6 clearly shows an increase of RT of the simulated 3rd party requests RT_{3rd} after the attack (blue line). The RT_{3rd} increased from around 90 ms on average to over 59 sec after the attack started. In total, the server was not able to respond to requests for a period of over 20 sec under the defined payload, resulting in a successful DoS attack.

V. Evaluation

The implemented DoS attacks were tested using four common Web service frameworks: Axis2 Java [21], Apache CXF [22], ASP.NET [25] and Metro [24]. Each of this framework was installed on the same Windows 7 machine (Core i5, 4GB RAM) using Java7 (Oracle 1.7.0_03). Additionally, we attacked the IBM DataPower XI50 [23], an XML Security Gateway, which comes with dedicated hardware. The attacked servers provided a simple echo service or were running one of its distributed sample services, which itself consumes no notable CPU/RAM. For the following evaluation, we left out the *SOAP Array* attack as the tested frameworks do not support this kind of data structure.

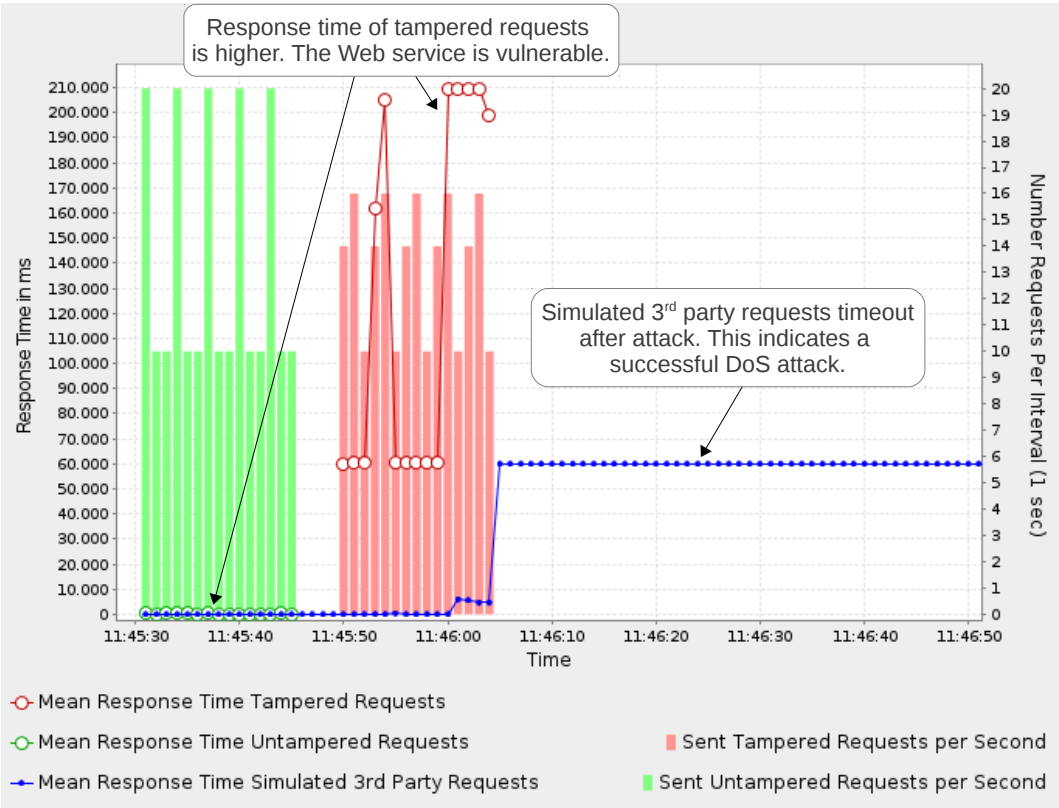

Fig. 6: Automatically generated results graph of a successful test on an vulnerable system

Attack name	Axis2 Java	Apache CXF	Metro	ASP.NET	XI50
Coercive Parsing	✓	✓	×	×	×
DJBX31A Hash Collision	✓	✓	✓	×	×
DJBX33A Hash Collision	×	×	×	×	×
DJBX33X Hash Collision	×	×	×	×	×
XML Attribute Count	✓	✓	✓	×	×
XML Element Count	×	✓	×	×	×
XML Entity Expansion	×	×	×	×	×
XML External Entity	×	×	×	×	×
XML Overlong Names	×	×	×	×	×

TABLE V: Vulnerability scan results.

The results of the vulnerability scan are shown in Figure V. Green values in Table VI denote a non-successful attack, red ones the opposite. The Axis2 Java and Apache CXF frameworks were vulnerable to the Coercive Parsing Attack, DJBX31A Hash Collision Attack[8] and XML Attribute Count Attack. Additionally, CXF was vulnerable to the XML Element Count Attack. Metro was only vulnerable to the DJBX31A Hash Collision[8] and the XML Attack Count attack. No vulnerabilities could be detected on the ASP.NET and the IBM XI50. For detecting these vulnerabilities, we tried to add the payload at different positions within the SOAP message, e.g. the XML Element Count Attack was only successful when placing the elements as child elements of the SOAP Header (and not within the SOAP Body).

[8]The reason for this is the used Java version, which is vulnerable to the DJBX31A Hash Collision Attack. To attack a server running a newer Java version, a different collision attack could be applied [16].

Attack name		Axis2 Java	Apache CXF	Metro
Coercive Parsing	ARTR	1952	1201	1.14
with 26,000 elements	RT [ms]	59000	40666	23
DJBX31A Collision	ARTR	10.48	3.47	5.09
with 4,500 attributes	RT [ms]	366	248	215
Attribute Count	ARTR	5.23	4.44	6.76
with 14,000 attributes	RT [ms]	145	155	130
Element Count	ARTR	1.33	7.69	1.07
with 45,000 elements	RT [ms]	55	236	21

TABLE VI: Attack impact presented using the mean of Attack-RT-ratio ($ARTR$) and Response Time (RT) values.

To clarify the impact of each vulnerability, Table VI shows the Attack-RT-ratio ($ARTR$) of each attack. Thereby, each attack was performed with a 180kB message size, where the attack impact was tried to be maximized, e.g. by using as many elements/attributes as possible (see Table VI). Aside, we added the mean of the time that was needed to process a tampered request. By Axis2 Java and Apache CXF the biggest damage caused the Coercive Parsing Attack. On the other

hand, application of this attack on Metro caused no DoS. DJBX31A Collision and Attribute Count attacks performed by the Metro framework similarly.

Note that these results were conducted only with non aggressive default settings – the message size could be increased drastically to get a higher impact. To turn this into a real attack, such that third party users are affected, the attacker has to increase the number of used threads and messages per second. This would also increase the $ARTR$ and processing time values.

Responsible Disclosure. We informed developers about these findings in February 2013. The Apache CXF developers applied a fix directly in the underlying Woodstox parser (version 4.2.0). They added ability to restrict certain size limits of parsed XML data. The other frameworks are currently being fixed.

VI. Conclusion

In this paper, we presented a custom Web service specific DoS penetration testing tool. The tool is built around a black-box test approach. For the first time, a tester needs no access to the tested target in order to evaluate attack success. The general design of the attack execution routine was illustrated and the design decisions were explained. Ten Web service specific DoS attacks were implemented and tested on common Web service frameworks. The results proved that the latest versions of the Axis2 Java, Apache CXF ans Metro frameworks are vulnerable to commonly known DoS attacks.

The future work will cover more Web service specific DoS implementations. XML Signature and XML Encryption based DoS attacks are yet to be implemented and tested. We hope that developers using this tool will get aware of these specific attacks and the tool will thus contribute to the deployment of more secure Web services.

Acknowledgements

We would like to thank the TÜVIT GmbH for giving the opportunity to test the tool on life systems. Moreover, we would like to thank Colm O hEigeartaigh for helpful discussions on this topic.

This research was partially supported by the Sec2 project of the German Federal Ministry of Education and Research (BMBF, FKZ: 01BY1030).

References

[1] T. Bray, J. Paoli, E. Maler, F. Yergeau, and C. M. Sperberg-McQueen, "Extensible markup language (XML) 1.0 (fifth edition)," W3C, W3C Recommendation, Nov. 2008. [Online]. Available: http://www.w3.org/TR/2008/REC-xml-20081126/

[2] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon, "Soap version 1.2 part 1: Messaging framework (second edition)," Tech. Rep., April 2007. [Online]. Available: http://www.w3.org/TR/soap12-part1/

[3] A. Nadalin, C. Kaler, R. Monzillo, and P. Hallam-Baker, "Web Services Security: SOAP Message Security 1.1 (WS-Security 2004)," *OASIS Standard*, 2006.

[4] S. Cantor, J. Kemp, R. Philpott, and E. Maler, "Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V2.0," OASIS Standard, 15.03.2005, 2005, http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf.

[5] A. S. Vedamuthu, D. Orchard, F. Hirsch, M. Hondo, P. Yendluri, T. Boubez, and U. Yalçinalp, "Web services policy 1.5 - framework," Tech. Rep., September 2007. [Online]. Available: http://www.w3.org/TR/ws-policy/

[6] N. Gruschka and L. Lo Iacono, "Vulnerable Cloud: SOAP Message Security Validation Revisited," in *ICWS '09: Proceedings of the IEEE International Conference on Web Services*. Los Angeles, USA: IEEE, 2009.

[7] J. Somorovsky, A. Mayer, J. Schwenk, M. Kampmann, and M. Jensen, "On breaking saml: Be whoever you want to be," in *21st USENIX Security Symposium*, Bellevue, WA, Aug. 2012.

[8] T. Jager and J. Somorovsky, "How To Break XML Encryption," in *The 18th ACM Conference on Computer and Communications Security (CCS)*, Oct. 2011.

[9] T. Jager, S. Schinzel, and J. Somorovsky, "Bleichenbacher's attack strikes again: breaking PKCS#1 v1.5 in XML Encryption." in *ESORICS*, ser. LNCS, S. Foresti and M. Yung, Eds. Springer, 2012.

[10] M. Jensen, N. Gruschka, and R. Herkenhöner, "A survey of attacks on web services," *Computer Science - R&D*, vol. 24, no. 4, pp. 185–197, 2009.

[11] B. Bosker, "Visa DOWN: WikiLeaks Supporters Take Down Site As 'Payback'," http://www.huffingtonpost.com/2010/12/08/visa-down-wikileaks-suppo_n_794039.html, accessed 01 July 2012.

[12] D. Pauli, "PayPal hit by DDoS attack after dropping Wikileaks," http://www.zdnet.com/news/paypal-hit-by-ddos-attack-after-dropping-wikileaks/489237, accessed 01 July 2012.

[13] F. Y. Rashid, "Anonymous Avenges Megaupload Shutdown With Attacks on FBI, Hollywood Websites," http://www.url.de/, accessed 01 July 2012.

[14] J. Wälde and A. Klink, "Hash Collision DOS Attacks," 28C3, http://www.nruns.com/_downloads/advisory28122011.pdf, Dec. 2011.

[15] S. A. Crosby and D. S. Wallach, "Denial of service via algorithmic complexity attacks," in *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, ser. SSYM'03. Berkeley, CA, USA: USENIX Association, 2003, pp. 3–3. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251353.1251356

[16] D. J. Bernstein, J.-P. Aumasson, and M. Boßlet, "Hash-Flooding DoS Reloaded: Attacks and Defenses," 29C3, Dec. 2012.

[17] M. Horsch and M. Stopczynski, "The German eCard-Strategy," 2011, technical Report.

[18] Kantara Initiative, "Kantara Initiative eGovernment Implementation Profile of SAML V2.0," June 2010, version 2.0.

[19] Danske Bank / Sampo Pankki, "Encryption, Signing and Compression in Financial Web Services," May 2010, version 2.4.1.

[20] C. Mainka, J. Somorovsky, and J. Schwenk, "Penetration testing tool for web services security," in *SERVICES Workshop on Security and Privacy Engineering*, Jun. 2012.

[21] Apache Software Foundation, "Apache Axis2," http://axis.apache.org/axis2/java/core.

[22] ——, "Apache CXF," http://cxf.apache.org.

[23] IBM, "WebSphere DataPower SOA Appliances," http://www-01.ibm.com/software/integration/datapower.

[24] The GlassFish Community, "Metro Web Service," http://metro.java.net/.

[25] Microsoft, "ASP.NET Web Services," http://msdn.microsoft.com/en-us/library/t745kdsh(v=vs.90).aspx.

[26] M. Vieira, N. Laranjeiro, and R. A. Oliveira, "Experimental Evaluation of Web Service Frameworks in the Presence of Security Attacks," June 2012.

[27] M. Jensen, N. Gruschka, and R. Herkenhöner, "A survey of attacks on web services," Springer-Verlag, 2009.

[28] mitre.org, "CAPEC-256: SOAP Array Overflow," http://capec.mitre.org/data/definitions/256.html, 2012.

[29] N/A, "Protecting Enterprise, SaaS & Cloud based Applications – A Comprehensive Threat model for REST, SOA and Web 2.0," Intel Corporation, Tech. Rep., 2009.

[30] A. Veithen, "Apache Axis2 Security Advisory - CVE-2010-1632," http://svn.apache.org/repos/asf/axis/axis2/java/core/security/CVE-2010-1632.pdf, 2010.

[31] N. Bhalla and S. Kazerooni, "Web Services Vulnerabilities," http://www.blackhat.com/presentations/bh-europe-07/Bhalla-Kazerooni/Whitepaper/bh-eu-07-bhalla-WP.pdf, Security Compass, February 2007, accessed 01 July 2010.