AdIDoS – Adaptive and Intelligent Fully-Automatic Detection of Denial-of-Service Weaknesses in Web Services

Christian Altmeier[1], Christian Mainka[2], Juraj Somorovsky[2], and Jörg Schwenk[2]

[1] Software AG
`Christian.Altmeier@softwareag.com`
[2] Horst Görtz Institute for IT Security
Ruhr University Bochum
{`christian.mainka,juraj.somorovsky,joerg.schwenk`}`@rub.de`

Abstract. Denial-of-Service (DoS) attacks aim to affect availability of applications. They can be executed using several techniques. Most of them are based upon a huge computing power that is used to send a large amount of messages to attacked applications, e.g. web services. Web services apply parsing technologies to process incoming XML messages. This enlarges the amount of attack vectors since attackers get new possibilities to abuse specific parser features and complex parsing techniques. Therefore, web service applications apply various countermeasures, including message length or XML element restrictions. These countermeasures make validations of web service robustness against DoS attacks complex and error prone.
In this paper, we present a novel adaptive and intelligent approach for testing web services. Our algorithm systematically increases the attack strength and evaluates its impact on a given web serice, using a blackbox approach based on server response times. This allows one to automatically detect message size limits or element count restrictions. We prove the practicability of our approach by implementing a new WS-Attacker plugin and detecting new DoS vulnerabilities in widely used web service implementations.

1 Introduction

Motivation. The principle of the Service Oriented Architecture (SOA) technology is to build a system of devices and machines connected via web services. A typical web service technology to establish SOA is SOAP [9]. SOAP-based web services are built upon the platform independent markup language – XML. They are used in business to business (B2B) process integrations and supported by large vendors like IBM and Axway.

The availability of web services in SOA scenarios is of huge importance. Therefore, Denial-of-Service (DoS) attacks on web services present a significant risk. DoS attacks attempt to exceed the consumption of computational resources, like CPU time or memory, with the goal that the system is no longer available for regular use. There are numerous techniques to perform DoS attacks, but throughout this paper, we concentrate on XML-based DoS attacks [4]. XML-based DoS is a special variant of a DoS attack that targets the XML parser. This means that the DoS payload is a specially crafted XML document, for example, a message with numerous deeply nested XML elements.

Complexity of XML-based DoS attacks. Previous research has revealed many different types of XML-based DoS techniques [4]. Unfortunately, the knowledge of these attacks is only the tip of the iceberg. The real challenge is to validate whether the tested XML parser is vulnerable to them. This is a complicated task, because there are many varieties of each single attack. For example, placing the DoS payload at one position within the XML document may affect the parser and result in a successful DoS attack, but using another position, for example a sibling element, can lead to an unaffected parser. Since there are many elements to place the payload in addition to many other aspects to consider, the detection of successful attack varieties is not trivial:

1. XML parsers can be configured to restrict a specific number of elements to be parsed. Consequently, attacks using more payload elements than this specific threshold will result in unsuccessful attacks. To detect such a threshold, the DoS attacks must be executed first with a small payload and then adaptively be adjusted due to the measured results.
2. The XML document structure can be validated using XML Schema [12]. Therefore, the attack payload cannot be placed at arbitrary positions in these scenarios. We use an approach that automatically reads the used XML Schema and places the payload at so-called *extension points* in such a way that the XML document containing the DoS payload is valid against the schema.

Contribution. In our work, we concentrate on the automatic detection of XML-based DoS and the automatic bypassing of countermeasures (XML Schema validation, thresholds, . . .). Our contributions are as follows:

– **AdIDoS** (Adaptive and Intelligent DoS), the first fully automatic XML-based DoS tool that detects DoS vulnerabilities with an intelligent and adaptive approach. Our tool extends the approach of [4], is open source, and part of WS-Attacker – a fully-automatic web service penetration testing framework.
– Our approach is **generic** and can be applied to XML scenarios beyond web services or even other DoS attacks beyond XML-based DoS[3].
– We evaluated **seven web service** implementations and give a detailed overview over their robustness against DoS attacks.

[3] Our implementation in the WS-Attacker framework is split into two parts: (1) a generic library to apply DoS attacks on XML and (2) a plugin that is used to transmit SOAP messages.

Outline. The following section will introduce the necessary foundations for this paper, including XML-based Web Services and XML-based DoS attacks. Section 3 spots on the complexity on evaluating DoS attacks. In Section 4, we elucidate the high-level design of our Adaptive Intelligent Denial-of-Service (AdIDoS) tool, while Section 5 gives more details on its implementation. We evaluate AdIDoS in Section 6, by testing five different web services and two XML security firewalls. We discuss related work in Section 7 and conclude in Section 8.

2 Foundations

In this section, we give a brief introduction to the relevant standards and technologies for this paper.

2.1 XML and XML Schema

eXtensible Markup Language (XML) is a structured format [2] by the World Wide Web Consortium (W3C), which allows transmission, validation and interpretation of data. The interpreted data can be described independently of software and hardware, thus XML is ideal to exchange data between different applications and organizations. The structure of an XML document is defined by XML elements. An XML element typically consists of a start tag `<tag>` and an end tag `</tag>`. It can include further *child* elements, element *attributes*, or text contents.

XML Schema is a recommendation by the W3C for describing the structure of an XML document [12]. It is basically a set of rules that can describe the structure for each contained element. It covers its allowed attributes, the type of its value (e.g., a string or integer), a description of its allowed child elements and how often they may occur.

2.2 Web Services

A web service is a method for interprocess interactions over networks between different software applications. A web service can be implemented using different technologies, for example, REST [5] or SOAP [9].

In this paper, we consider the SOAP technology. SOAP (originally defined as Simple Object Access Protocol) is a W3C specification defining the structure of XML messages and a protocol to achieve a machine-to-machine communication. SOAP messages generally consist of *header* and *body*. The `<Header>` element includes message-specific data (e.g. timestamp, user information, or security tokens). The `<Body>` element contains function invocation data.

2.3 XML-based DoS Attacks

There are numerous XML-based DoS attacks. In the next section, we will give a more detailed description of the *Coercive Parsing Attack* and use this attack as a running-example in the following sections through this paper.

Coercive Parsing Attack. The Coercive Parsing attack creates a deeply nested XML document. If the document is parsed by a vulnerable service, memory exhaustion occurs. The following SOAP message gives an example with deeply nested elements.

```
<soap:Envelope xmlns:soap="...">
  <soap:Header></soap:Header>
  <soap:Body>
    <x>
      <x>
        <!-- deeply nested -->
      </x>
    </x>
  </soap:Body>
</soap:Envelope>
```

This is only one example for a Coercive Parsing attack. It is also possible to place the payload (the `<x>`-elements) in other elements, for example inside the `<Header>` element. Additionally, one can vary the number of nested elements. All these aspects affect the impact and the success-level of the attack.

Further XML-based DoS Attacks. The following XML-based DoS attacks are described in [4] and also implemented in AdIDoS:

- Coercive Parsing Attack
- XML Element Count Attack
- XML Attribute Count Attack
- XML Entity Expansion Attack
- XML External Entity Attack
- XML Overlong Names Attack
- HashCollision Attack

2.4 Attack Roundtrip Time Ratio (ARTR)

Our automatic tool AdIDoS evaluates the effectiveness of different DoS attacks against a server in a black-box manner. Thus, the only measurable metric is *time.*

We define the time of the last byte sent by a client's request up to the time of the first byte of the corresponding response as the roundtrip time:

$$RT = t_{\text{Received}} - t_{\text{Sent}}$$

If a request does not contain a DoS payload, we refer to it as an *untampered* request. Consequently, a request with DoS payload is referred as a *tampered request.*

We use the Attack Roundtrip Time Ratio (ARTR) [4] as a metric to measure the impact of each DoS attack variant and to be able to compare them. ARTR is defined as the quotient of the roundtrip time of tampered and untampered requests [4]. The higher the ARTR value is, the more effective is the attack.

2.5 WS-Attacker

WS-Attacker is a modular framework for web services penetration testing [8], available as an open source project on Github.[4] WS-Attacker uses a plugin architecture to execute XML-specific attacks on web services automatically. In its current version, WS-Attacker supports the following attacks:

1. SOAPAction Spoofing [8].
2. WS-Addressing Spoofing [8].
3. Basic XML Denial-of-Service Attacks [4].
4. XML Signature Wrapping [3].
5. Attacks on XML Encryption [7].

In this paper, we extend the functionality of WS-Attacker and implement AdIDoS as an attack plugin.

3 DoS Complexity

The complexity of DoS attacks is founded in two parts:

1. Since we assume to have *no physical access*, we can only perform black-box tests.
2. Our DoS attacks abuse weaknesses in XML parsers. As such, we need to make use of the XML document structure.

3.1 Black-box Tests

A black-box penetration test refers to a methodology of testing a computer system without knowledge of its internals. Therefore, we do not have the possibility to measure the CPU load or memory consumption of the tested service. We only rely on the ARTR: We measure and evaluate the time that the service needs to process the request and compute the response, including the network transfer time.

3.2 XML Document Structure

XML-based DoS abuses weaknesses in the underlying XML parser. Each XML parser has its own behavior and can be adjusted to fit the service's requirements. This increases the complexity to apply a DoS attack dramatically. Some XML parserss:

1. Only process unexpected elements, if they are placed to a specific position in the XML document (they validate the XML Schema).
2. Only allow a specific number of elements or attributes in an XML document (thresholds).

[4] `https://github.com/RUB-NDS/WS-Attacker`

Additionally, the service itself may restrict the amount of request by one client within a time period.

XML Schema. If an XML Schema validation is performed by the XML parser, placing the payload of an XML-based DoS attack at a specific position inside the document can be detected. The attack will not succeed. Taking the example of the Coercive Parsing attack, placing the `<x>` elements as a child of the `<soap:Envelope>` element would break the SOAP schema. However, placing the same `<x>` elements as a child of the `<soap:Header>` element is conform to the SOAP schema and the message will be accepted — the attack can potentially be applied.

In addition to the above behavior, some parsers skip specific document parts. For example, a web service that does not use *any* SOAP extensions could skip to parse the whole `<soap:Header>` element and continue with the `<soap:Body>`. This means that placing the XML-based DoS payload in the header does not result in a successful attack, while placing it in the body could.

Thresholds. Some XML parsers implement thresholds. They stop to process incoming messages if they parse more than a specific number of elements, attributes, or bytes.

Suppose an XML parser that only accepts messages up to 100 elements. Applying a Coercive Parsing attack with 5000 nested elements will result in an unsuccessful attack, but the implementation could be vulnerable if the attack is applied with, for example, 80 nested elements. To make the attack detection more accurate, it is important to start XML-based DoS attacks with a small payload, and increase it by time. This way, possible thresholds can be detected.

4 Design

In this section, we describe several principles and design decisions we followed in order to create the adaptive and intelligent XML-based DoS attack plugin AdIDoS.

4.1 Automatic DoS Detection Workflow

AdIDoS systematically tests the web service for DoS weaknesses. The detection workflow is fully automatic and AdIDoS uses the following algorithm to proceed (see Figure 1):

1. AdIDoS chooses one attack from its pool of implemented DoS attacks.[5]
2. It specifies the position where to set the payload. Therefore, XML Schema is used to determine all matching positions.

[5] Its current implementation includes *Coercive Parsing*, *XML Attribute Count*, *XML Element Count*, *XML Entity Expansion*, *XML External Entity*, *XML Overlong Names*, and 4 variants of *HashCollision* attacks – 10 attack variants in total.

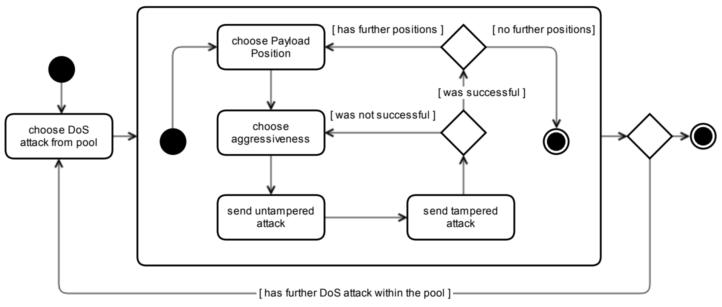


Fig. 1. AdIDoS simplified workflow of systematic DoS detection

3. The *aggressiveness* of the attack is specified. Aggressiveness means, how much XML payload is responsible for the attack. The more XML payload the attack uses, the more aggressive it is. For example, a coercive parsing attack using the payload `<x><x></x></x>` is more aggressive than an attack with `<x></x>`. Each attack variant will start with very low aggressiveness and adjust it depending on the ARTR.
4. The algorithm generates an untampered request and executes the attack against the web service. This information is used as a base line for the later decision, whether an attack is successful or not.
5. It generates a tampered request and executes the attack against the target web service.
6. It analyzes the roundtrip time of the untampered and tampered requests and decides whether the attack is successful or not by computing the ARTR (See Section 5.3 for details).
   – Successful: the attack is marked as successful for this payload position, the next position is specified, followed by Step 3.
   – Not successful: a more aggressive attack is set, followed by Step 4.
   This step is performed as long as further parameter sets are available for the DoS attack. Hereafter the next DoS attack is chosen and AdIDoS continues with Step 1.

4.2 Automatic Threshold Detection

The most effective countermeasure against XML-based attacks is to limit the number of elements/attributes which can occur in an XML document, or the size of the document. In our approach, we automatically detect and narrow down thresholds used by a web service. Thereby a variation of the binary search algorithm is used, which is shown in Figure 2. The steps are as follows:

1. The threshold detection is initialized with the weakest and the strongest attack vector. The weakest vector (our minimum) is the least aggressive attack that was executed successfully. The strongest vector (our maximum) is the attack vector that was not successful.

2. The strength of each newly created attack vector is generated as an average of the weakest and the strongest aggressive attack vector.
3. The relevant information in this phase is the execution state of the attack:
   – Successful: the current attack vector strength is set as a new minimum
   – Not successful: the current attack vector strength is set as new maximum

Depending on the expected precision, these steps can be repeated several times. In our measurements, five iterations showed up to provide values giving enough information about the analyzed web service. The detected threshold is then stored in memory and can be considered for further web service investigation by the developer.

After the process of narrowing down the threshold, AdIDoS returns to its normal analysis but will now consider the detected threshold. In addition, the web service is tested for DoS weaknesses near the actual threshold.

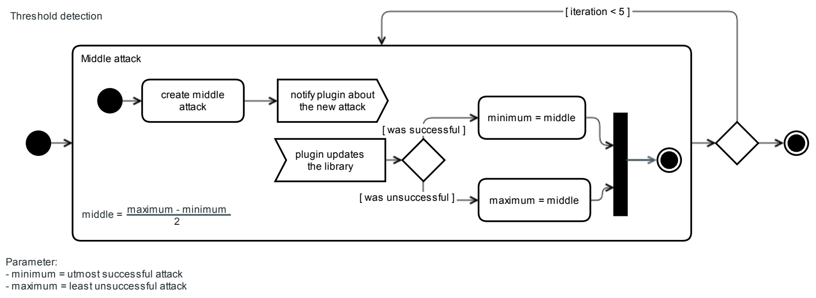

Threshold detection
[iteration < 5]
Middle attack
create middle attack
notify plugin about the new attack
plugin updates the library
[was successful]
[was unsuccessful]
minimum = middle
maximum = middle
middle = maximum - minimum / 2
Parameter:
- minimum = utmost successful attack
- maximum = least unsuccessful attack

Fig. 2. Threshold detection

5 Implementation

We implemented the concepts described in the previous section as a WS-Attacker plugin AdIDoS. In the following, we give a detailed view on some specific implementation issues.

5.1 AdIDoS for WS-Attacker

We implemented all XML-based DoS attacks listed in Section 2.3 as a WS-Attacker plugin – called AdIDoS (Adaptive Intelligent Denial-of-Service). Each DoS attack executed by AdIDoS is a composition of multiple parameters. There are two types of attack parameters:

– **Independent** attack parameters are generic configuration parameters which can be used for all DoS attacks. Example for independent parameters are the number of used threads to send requests, or the delay between sending them.

– **Dependent** attack parameters are specific for the executed DoS attack. For example, in Coercive Parsing, AdIDoS chooses the number of nested elements.

In addition to that, there are multiple possibilities for placing the attack payload (e.g. in the `<soap:Header>`, in the `<soap:Body>`, ...). All positions are marked in the XML message by analyzing its XML Schema. We used XML Schema parsing to automatically detect so called XML extension points.[6] These extension points can be used to place the payload without invalidating the schema. If the web service uses XML Schema validation, our generated attack messages do not harm the schema. Every DoS attack specifies where its payload can be placed:

– `ELEMENT`: the payload of an attack can be placed as a new element into the document
Supported by Coercive Parsing, XML Element Count, XML Entity Expansion, XML External Entity and XML Overlong Names
– `ATTRIBUTE`: the payload can be placed within an existing element
Supported by XML Attribute Count and HashCollision

5.2 Attack Configuration and Execution

By executing a concrete DoS attack, AdIDoS first uses the schema analyzer provided by WS-Attacker to identify all available extension points. Hereafter AdIDoS provides a pool of various XML-based DoS attacks, which can easily be configured through the configuration dialog as shown in Figure 3 on the left. This is extremely useful if the tester just wants to execute a subset of the supported attacks to save time. Every attack has its own set of supported parameters. Figure 3 shows the attack parameters for the *Coercive Parsing Attack* on the right. *Coercive Parsing* uses two parameters:

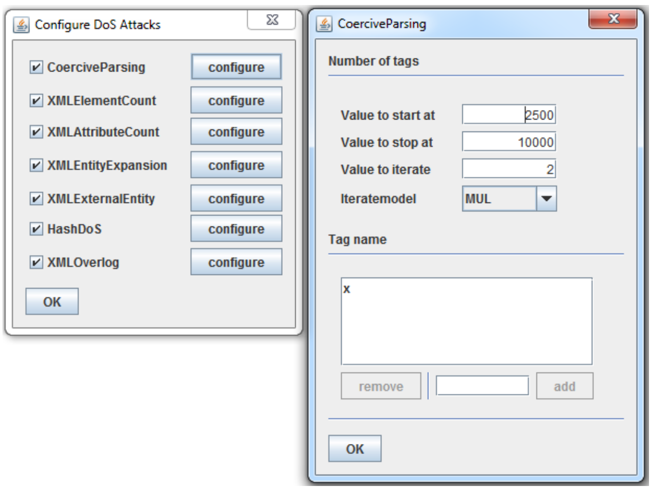
– *Number of tags*: For this parameter a range of values can be specified. In addition, the step size can be set.
– *Tag name*: This parameter can be specified as a list of values.

The range option allows one to perform attacks with various levels of aggressiveness.

5.3 Attack Success and Efficiency Decision

The success of an attack is calculated as follows: We use the median round trip time of untampered requests in comparison to the median round trip time of tampered requests. To compute the median, we use the last ten (untampered

[6] Areas in the XML document, where additional elements or attributes can be placed according to the schema definition. Identified by `<xs:any>` and `<xs:anyAttribute>` in the XML Schema


Configure DoS Attacks

- ✓ CoerciveParsing configure
- ✓ XMLElementCount configure
- ✓ XMLAttributeCount configure
- ✓ XMLEntityExpansion configure
- ✓ XMLExternalEntity configure
- ✓ HashDoS configure
- ✓ XMLOverlog configure

OK

CoerciveParsing

Number of tags

Value to start at	2500
Value to stop at	10000
Value to iterate	2
Iteratemodel	MUL

Tag name

x

remove add

OK

Fig. 3. Configuration of Denial-of-Service attacks

or tampered) requests sent to the web service. If the median round trip time of the tampered requests is three times higher[7] than the median round trip time of the untampered requests, the attack is marked as successful. It allows one to reliably recognize attacks as successful and minimizes the false positives.[8] The attack success is recognized as follows:

- *ratio time* <*3*: the attack was not successful
- *ratio time* >= *3*: the attack was successful

Besides the information that an attack is successful, AdIDoS also provides an estimation of the attack efficiency. Again this estimation is based on the median round trip time of the two attack runs.

- *ratio time between* >= *3 and* <*6*: the attack was efficient
- *ratio time* >= *6*: the attack was highly efficient

To avoid false positives, the AdIDoS algorithm uses an approach with a single success confirmation. If the algorithm detects measurable differences between tampered and untampered roundtrip times, the server first gets some time to recover. This prevents that a DoS attack is marked as successful even though it is not, just because it is executed right after a successful attack. After the recovery time, a new attack vector is sent to the server and its response time is compared to the response times of untampered requests.

5.4 Extended ARTR Approach

Falkenberg et al. [4] presented an algorithm for attack success measurements that uses a blackbox approach with an ARTR metric (see Section 2.4). Their

[7] This value was chosen empirically based on our tests in local networks.
[8] Here an attack is marked as successful even though is is not

ARTR approach was based on measuring response times. The response time measurement always started with the first byte that was sent, and stopped with the last byte that was received. With this algorithm the comparison of two or more requests requires that the requests must have the same size. Otherwise the transmission of the data would affect the measurement.

AdIDoS also pursues a blackbox approach with the ARTR metric. However, it uses a slightly different measurement algorithm (see Figure 4). The response time measurement starts with the last byte sent, and stops with the first byte received. The main benefit of this improved time measurement is that fluctuations, which can occur during transfer, do not affect the measurement as strongly as before. In addition, only the time is measured that the service needs to execute the request. Finally, it becomes less important to send requests of the same size.

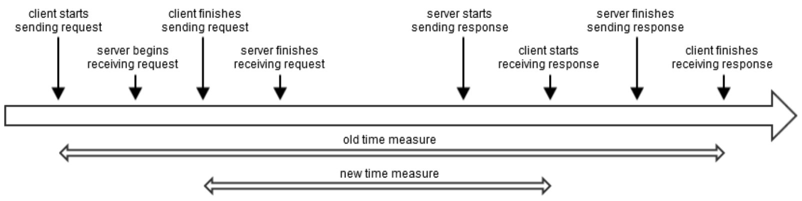

client starts sending request
server begins receiving request
client finishes sending request
server finishes receiving request
server starts sending response
client starts receiving response
server finishes sending response
client finishes receiving response
old time measure
new time measure

Fig. 4. Our new ARTR approach considers only time between the last byte that was sent, and the first byte that was received.

6 Practical Evaluation

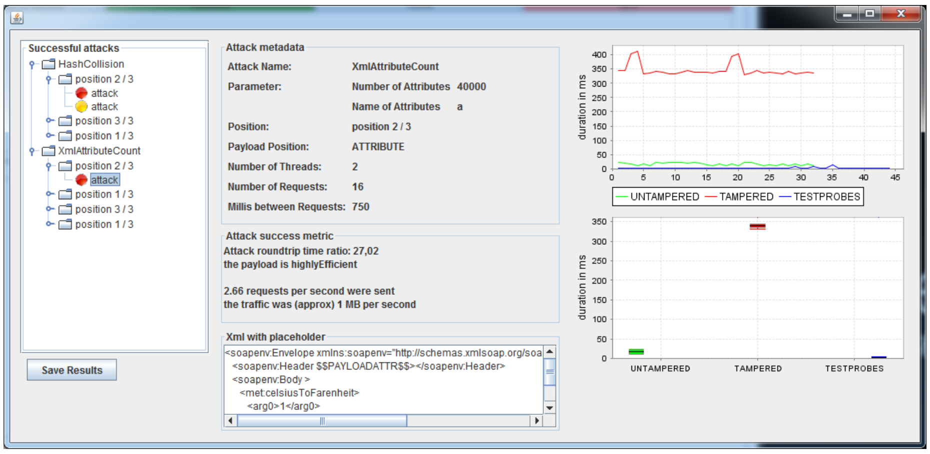
Using AdIDoS, it becomes easy to test a given web service for DoS weaknesses. Multiple test scenarios were set up to investigate common web service frameworks: Apache Axis2 [13], Apache CXF [14], Metro [15], .NET [10] and PHP [16].

The services were hosted on a Windows 7 machine (@2,30 GHz, 4GB Ram) with the following set up:

– Java based web services: Tomcat 7.0.55 (Oracle Java7 1.7.0_71)
– .NET: IIS 7.5.7600.16385 (.NET framework v2.0.50727)
– PHP: Apache 2.4.12 (PHP 5.5.24.0)

The tests were performed from a second, independent Windows 7 machine within the same LAN with the default configuration and parameters. As a service a simple conversion service was implemented, which converts Fahrenheit to Celsius and vice versa. In addition, the XML Security Gateways WebSphere DataPower Integration Appliance XI50 [6] and Axway SOA Gateway 7.3.1 [1] were tested.

Table 1 gives an overview of all tested web services. Apache CXF version was the most secure open source web service service framework. It was the only



Attack metadata

Attack Name:	XmlAttributeCount
Parameter:	Number of Attributes 40000
	Name of Attributes a
Position:	position 2 / 3
Payload Position:	ATTRIBUTE
Number of Threads:	2
Number of Requests:	16
Millis between Requests:	750

Attack success metric

Attack roundtrip time ratio: 27,02
the payload is highlyEfficient

2.66 requests per second were sent
the traffic was (approx) 1 MB per second

Xml with placeholder

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soa
  <soapenv:Header $$PAYLOADATTR$$></soapenv:Header>
  <soapenv:Body >
    <met:celsiusToFarenheit>
      <arg0>1</arg0>
```

Fig. 5. Automatically generated result view of successful attacks with concrete information.

	Apache Axis2	Apache CXF	Metro	.NET	PHP	XI50	Axway
Coercive Parsing	⚡	-	-	-	-	-	-
XML Element Count	-	-	-	-	-	-	-
XML Attribute Count	⚡	-	⚡	⚡	⚡	⚡	-
XML Entity Expansion	-	-	-	-	-	-	-
XML External Entity	-	-	-	-	-	-	-
HashCollision	⚡	-	⚡	-	⚡	-	-
XML Overlong Names	-	-	-	-	-	-	-

Table 1. Results of our vulnerability scan. The Symbol "⚡" marks web services, where DoS weakness were found by AdIDoS.

open source framework that provides a secure default configuration. The CXF implementation limits the possible appearance of elements in an XML document to achieve this goal.
The Apache Axis2 framework is vulnerable to Coervice Parsing, XML Attribute Count and HashCollision with the collision generators DJBX31A and DJBX33A. It is very unusual that one implementation is vulnerable to multiple collision generators, and we cannot explain this behavior. The vulnerability to Coercive Parsing and XML Attribute Count (on ELEMENT) is limited to the `soap:Header`. This indicates that unexpected elements are only processed at this position. The highest impact comes from XML Attribute Count, only CXF was not vulnerable to this attack.

In contrast to the expected behavior of the two security gateways, the XI50 was also vulnerable to XML Attribute Count. By placing the attack payload within an existing element in the `soap:Body` there was a clear evidence for a higher processing time.

Threshold for	Apache CXF	XI50	Axway
Nested Elements	80 - 158	470 - 548	236 - 314
Number of Elements	-	-	783 - 1,173
Number of Attributes	626 - 704	-	704 - 782
Element name length	-	3,125 - 3,515	3,906 - 4,296
Attribute length	116,226 - 122,343	-	-
Number of Entities	-	-	16 - 32

Table 2. Overview of thresholds used in the tested frameworks.

Besides the detection of DoS weaknesses, AdIDoS is able detect thresholds used by the implementations. These thresholds are considered for further investigation of a service. Table 2 shows the detected thresholds and their approximate value.

Attack name		Axis2	Metro	.Net	PHP
Coercive Parsing	ARTR	6.52			
	Number of Tags	2,500			
XML Attribute Count	ARTR	4.02	7.00	3.30	10.65
	Number of Attributes	10,000	10,000	10,000	10,000
HashCollision	ARTR	12.75	6.21		155.88
	Number of Collisions	3,750	3,750		1,250

Table 3. Average ARTR and attack parameters.

AdIDoS performs multiple attacks against a web service. The impact of an attack is shown by ARTR and the used parameters. Table 3 illustrates the ARTR for the tested web services and Table 4 illustrates the ARTR for the XI50 security gateway. Beside the ARTR the used parameters for the single attacks are specified.

Attack name		XI50
XML Attribute Count	ARTR	7.79
	Number of Attributes	2,500

Table 4. Average ARTR and attack parameters for XI50

The goal of AdIDoS is to detect DoS weaknesses in XML-based web services and not to exploit them. For this reason, AdIDoS stops as soon as a DoS weakness for an attack class (e.g. Coercive Parsing) is detected. More aggressive attacks, which certainly result in a higher ARTR, are not performed.

7 Related Work

There are already DoS attacks that rely on handling of XML data. These attacks are partially supported by penetration testing tools like SoapUI,[9] or WS-Fuzzer.[10] SoapUI and WSFuzzer are tools developed specifically for testing web service platforms, but these tools have no support for automatic XML-based DoS analysis.

Oliveira et al. implemented a web service tool called WSFAggressor [17], which contains several 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 [6], or pen-testing sensitive customers' servers. Moreover, this tool misses some important attack techniques such as HashDoS [18].

Falkenberg et al. studied XML-based DoS attacks [4] and implemented a WS-Attacker DoS plugin. The plugin does not need access to the tested web service in order to measure the attack success. It instead uses a blackbox approach using the server response times (ARTR) only. In contrast to AdIDoS, the authors do not analyze an adaptive approach of XML-based DoS testing: Values and size of tampered messages is chosen statically, and the penetration tester has to adapt these properties manually. This results in attack testing complexity and to possible false negatives. In our work, we extended the approach of Falkenberg et al. and implemented an adaptive and intelligent detection XML-based DoS attacks.

Very recently, Pellegrino et al. studied data compression attacks against several applications [11], including web services servers. In order to execute an attack against a web service server, the attacker inserts a huge number of spaces into a SOAP message and compresses the message using a deflate algorithm (used by zlib, gzip or zip libraries). This way, a compression ratio of about 1:1000 can be achieved. The authors reported that Apache Axis2 and Apache CXF were vulnerable to these attacks. These attacks are currently missing in WS-Attacker and can be implemented in a future work.

8 Conclusions and Future Work

In this paper, we developed a new approach for testing robustness of XML-based web services against DoS attacks. Our approach adapts an intelligent strategy that automatically increases the attack strength and searches for attack

[9] `http://www.soapui.org`

[10] `http://sourceforge.net/projects/wsfuzzer`

thresholds. We implemented the approach as a new plugin for the web service penetration testing framework WS-Attacker. Interestingly, the plugin allowed us to detect new attacks, previously overlooked in related works. This proves the feasibility of our new approach for testing DoS attacks.

While our paper investigates SOAP-based web services, the implemented library can be directly applied to further XML standards as well, e.g. SAML or REST-based web services. Moreover, the general idea of intelligent DoS testing can be adapted to other applications beyond XML as well.

Further research in this direction could be in extending the number of web service specific attacks. As described in [4,11], further attacks like Recursive Cryptography, XML Signature Key Retrieval DoS, or data compression attacks are applicable to web services as well.

The values for detection of attack success and efficiency were chosen empirically based on our observations in local networks. However, different network conditions could affect the results and introduce new false positives and false negatives. In order to detect DoS attacks over the Internet, the accuracy of our solution has to be improved.

Acknowledgements

We would like to thank our anonymous reviewers for their helpful comments. The research was supported by the *German Ministry of research and Education (BMBF)* as part of the `VERTRAG` research project.

References

1. Axway: Axway soa gateway, `https://www.axway.com/products-solutions/soa-governance/soa-gateway`
2. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F.: Extensible markup language (xml) 1.0 (fifth edition) (November 2008), `http://www.w3.org/TR/REC-xml/`
3. Christian Mainka: Automatic Penetration Test Tool for Detection of XML Signature Wrapping Attacks in Web Services (May 2012), Master thesis supervised by Jörg Schwenk and Juraj Somorovsky
4. Falkenberg, A., Mainka, C., Somorovsky, J., Schwenk, J.: A New Approach towards DoS Penetration Testing on Web Services. In: Web Services (ICWS), 2013 IEEE 20th International Conference on. pp. 491–498. IEEE (2013), `http://dblp.uni-trier.de/db/conf/icws/icws2013.html#FalkenbergMSS13`
5. Fielding, R.T., Taylor, R.N.: Principled design of the modern web architecture. ACM Trans. Internet Technol. 2(2), 115–150 (May 2002), `http://doi.acm.org/10.1145/514183.514185`
6. IBM: Websphere datapower integration appliance xi50, `https://www-03.ibm.com/software/products/en/datapower-xi50`
7. Kupser, D., Mainka, C., Somorovsky, J., Schwenk, J.: How to break xml encryption – automatically. In: 9th USENIX Workshop on Offensive Technologies (WOOT 15). USENIX Association, Washington, D.C. (Aug 2015), `https://www.usenix.org/conference/woot15/workshop-program/presentation/kupser`

8. Mainka, C., Somorovsky, J., Schwenk, J.: Penetration testing tool for web services security. In: SERVICES Workshop on Security and Privacy Engineering (Jun 2012)
9. McCabe, F., Booth, D., Ferris, C., Orchard, D., Champion, M., Newcomer, E., Haas, H.: Web services architecture. W3C note, W3C (Feb 2004), http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/
10. Microsoft: .net framework, `https://msdn.microsoft.com/en-us/library/a4t23ktk(v=vs.80).aspx`
11. Pellegrino, G., Balzarotti, D., Winter, S., Suri, N.: In the compression hornet's nest: A security study of data compression in network services. In: 24th USENIX Security Symposium (USENIX Security 15). pp. 801–816. USENIX Association, Washington, D.C. (Aug 2015), `http://blogs.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/pellegrino`
12. Sperberg-McQueen, C.M., Thompson, H.S., Maloney, M., Thompson, H.S., Beech, D., Mendelsohn, N., Gao, S.S.: W3C xml schema definition language (XSD) 1.1 part 1: Structures. Last call WD, W3C (Dec 2009), `http://www.w3.org/TR/2009/WD-xmlschema11-1-20091203/`
13. The Apache Software Foundation: Apache axis2, `https://axis.apache.org/axis2/java/core/`
14. The Apache Software Foundation: Apache cxf – index, `https://cxf.apache.org/`
15. The GlassFish community: Metro, `https://cxf.apache.org/`
16. The PHP Group: Php: Hypertext preprocessor, `https://php.net`
17. Vieira, M., Laranjeiro, N., Oliveira, R.A.: Experimental Evaluation of Web Service Frameworks in the Presence of Security Attacks (June 2012)
18. Wälde, J., Klink, A.: Hash Collision DOS Attacks. 28C3, `http://www.nruns.com/_downloads/advisory28122011.pdf` (Dec 2011)