

Protecting Web Services from DoS Attacks by SOAP Message Validation

Nils Gruschka, Norbert Luttenberger

Dept. for Computer Science
Christian-Albrechts-University of Kiel
{ngr|nl}@informatik.uni-kiel.de

Abstract. Though Web Services become more and more popular, not only inside closed intranets but also for inter-enterprise communications, few efforts have been made so far to secure a Web Service's availability. Existing security standards like e.g. WS-Security only address message integrity and confidentiality, and user authentication and authorization. In this article we present a system for protecting Web Services from Denial-of-Service (DoS) attacks. DoS attacks often rely on malformed and/or overly long messages that engage a server in resource-consuming computations. Therefore, a suitable means to prevent such kinds of attacks is the full grammatical validation of messages by an application level gateway before forwarding them to the server. We discuss specific kinds of DoS attacks against Web Services, show how message grammars can automatically be derived from formal Web Service descriptions (written in the Web Service Description Language), and present an application level gateway solution called "Checkway" that uses these grammars to filter Web service messages. The paper closes by giving some performance figures for full grammatical validation.

1 Introduction

As Web Services become more and more popular, not only inside closed intranets but also for inter-enterprise communications, security is becoming crucial for operating Web Services. While the basic Web Service specifications ([4], [10]) themselves do not address any security topics, a large number of additional specifications (WS-Security [3], WS-SecurityPolicy [6], WS-Trust [12], WS-SecureConversation [11] etc.) for Web Services security exists. However all these standards focus on the aspects of message *integrity* and *confidentiality* and user *authentication* and *authorization*.

Few efforts have been made so far to secure the Web Service server itself and ensure a Web Service's *availability*. Of course traditional perimeter protection systems like packet filters, application level gateways, and intrusion detection systems contribute to this, but we will show that these are unable to secure a Web Service server's availability in an adequate manner.

In this article we present an application level gateway system for protecting Web Services from Denial-of-Service (DoS) attacks. DoS attacks often rely

on malformed and/or overly long messages that engage a server in resource-consuming computations. For Web Services a suitable means to prevent such kinds of attacks is the full grammatical validation of messages by an application level gateway before forwarding them to the server. Web Service messages are XML documents and these are usually defined by an XML Schema, written in the XML Schema definition language—a grammar language for XML. Our system generates an XML Schema from a Web Service description and validates all Web Service messages against this schema.

This article is organized as follows: The next chapter introduces Denial of Service attacks in general and in the context of Web Services. Chapter 3 discusses the protection of Web Services from DoS attacks and introduces our solution. In chapter 4 the processing of a Web Service description for our Web Service firewall and in chapter 5 the firewall itself are presented. The article closes with an outlook.

2 Attacks on Services

Denial of Service (DoS) attacks aim at reducing or completely eliminating a systems's or service's availability. One can distinguish two kinds of DoS attacks: *Protocol Deviation Attacks* and *Resource Exhaustion* [17]. Protocol Deviation Attacks exploit vulnerabilities in implementations of protocol processing entities. In some cases a single packet that diverges from the intended protocol flow can make the attacked system crash. A well-known example is *Ping of Death*.

Resource Exhaustion attacks consume the resources necessary to provide the service (network bandwidth, memory and computation resources). The simplest attack produces an extremely high network traffic load to the system providing the service (*Dump Flooding*). Using such an attack makes it difficult to completely interrupt a service's availability, even if executed as a *Distributed Denial of Service* (DDoS) attack. More elaborated DoS attacks do not try to occupy all available network capacity by brute force, but send messages that—though comparably small in number—are suited to quickly exhaust the server's memory and cpu resources. A popular example is the *TCP/SYN flooding*, where the server is flooded with (small) TCP/SYN packets. The server must create a complete TCP connection context for each packet and finally crashes due to memory consumption.

With XML and Web Services new kind of attacks arise. The most common message protocol for Web Services is SOAP, an XML based message format. Such a SOAP message is usually transported using the HTTP protocol. Figure 1 shows a simple SOAP message with the most relevant HTTP header lines. The message contains a request for the operation `add` with 3 parameters named `x`.

Two of the most important DoS attacks on XML based services like Web Services are *Coercive Parsing* and *Oversize Payload* (see e.g. [18] and [13]). The first one uses a deeply nested XML document, the second one an extremely large XML document to exhaust the service's memory. This is easier than for non-XML protocols due to the nature of XML document processing. An incoming

```

POST /WebServices/MathService.asmx HTTP/1.1
SOAPAction: "http://example.com/add"
Content-Type: text/xml

<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:ns="http://example.com/AddService">
  <soap:Body>
    <ns:add>
      <ns:x>12</ns:x>
      <ns:x>38</ns:x>
      <ns:x>27</ns:x>
    </ns:add>
  </soap:Body>
</soap:Envelope>

```

Fig. 1. Sample SOAP message

SOAP message is parsed, validated to the Web Service interface specification and bound to programming language objects [14]. The most common and flexible model for XML processing is DOM [2]. When using DOM a DoS attack would indeed be very simple. A DOM based parser reads the complete SOAP message and builds an in-memory representation (called DOM tree), that is much larger than the message itself. The parser can therefore be attacked by an arbitrary SOAP message, e.g. a message with a large total size or with a deeply nested XML structure.

Even if the the parser component of the Web Service uses an event-based processing model (e.g. SAX [15]) and the succeeding components (for validating and language binding) check the correctness of the SOAP message, there are still possibilities for attacks. A simple, yet effective attack can be performed if the Web Service message contains a list of elements (like in the sample shown in figure 1). This is defined in the Web Service interface description (see section 4.1) by an XML Schema element [7] containing an attribute `maxOccurs="number-of-elements"`. If this element has a cardinality > 1 , the number of elements is nearly always set to "unbounded" to simplify the Web Service processing. If the description is generated by a Web Service framework from an existing implementation (which is a very common proceeding), this value is automatically set for all data arrays. Such a declaration allows documents to contain an unlimited number of elements. It is obvious that such a document can exhaust the server's memory. In practical tests we easily produced deadlocks and crashes, sending a SOAP message with a large number of elements to a .NET and an AXIS Web Service.

Though not a DoS attack, a further important attack on Web Services covered by our solution is *WSDL Scanning*. All operations for a service are described and advertised inside a Web Service description using the *Web Service Description*

Language (see section 4.1). If only some of a service's operations are intended to be called from the internet, an attacker is able to call all the service's operations anyway. Packet Filters and HTTP ALG are unable to differentiate operations belonging to the same service, because they all have the same service endpoint (IP address, TCP port and HTTP URL). The operation is only defined inside the SOAP message¹ (e.g. the operation `add` in the SOAP message in figure 1).

3 Protecting Web Services

3.1 Web Services and Firewalls

Today it is common practice that hosts and services inside a private network (whether enterprise or home) are protected by a firewall system². The firewall has two tasks: 1. to protect the services from attacks and 2. to prevent access to services, which shall not be reachable from the internet.

The most widespread firewall concept is packet filtering. Packet filters operate on layer 3 and 4 of the ISO/OSI Basic Reference Model and analyse IP and TCP headers. Such firewalls are suitable for protecting against DoS attacks exploiting the TCP or IP protocol, like Ping of Death or the TCP/SYN Flood. It is also capable to filter accesses to services using the target IP address and the target TCP port.

Application level gateways (ALG) are defined to analyse application level protocols above ISO/OSI layer 4. Actual ALGs understand simple application protocols like HTTP. Such a HTTP ALG protects a service from attacks using malformed HTTP requests and attacks like Cookie Poisoning. It can limit access to services using the HTTP request URL.

But how can Web Services be protected? Packet filters and HTTP ALGs only check the TCP, IP and HTTP protocol header, but not the SOAP message.

A Web Service defines the valid SOAP messages using a Web Service interface description (see section 4.1). Processing of SOAP messages is time and memory consuming for the Web Service server, so every *non-valid* message should be rejected by a Web Service firewall. This can be done by validating the SOAP message in an external application level gateway.

A very simple countermeasure against large *valid* messages is limiting the SOAP message's total size. This can even be done by a simple firewall without checking the SOAP messages itself. On the other hand, this is not very sensible. The amount of memory needed while processing an XML document is usually much larger than the document itself. In order to avoid attacks, the size limit should be low. Unfortunately, this could exclude many valid documents.

¹ The HTTP header field `SOAPAction` also includes the operation, but this is only a hint to the actual Web Service operation inside the SOAP message and should not be taken into account (see also [9], 4.3.4)

² The term *firewall* is often used for packet filtering systems, that analyse only IP addresses and TCP ports. In this article we use *firewall* generalised for all security systems, which analyse and filter data traffic

look at Web Service client/server interaction and the Web Service interface description are required.

4 Web Service Interface Description

4.1 WSDL Structure

The Web Service interface description is composed using the *Web Service Description Language* (WSDL) [5].

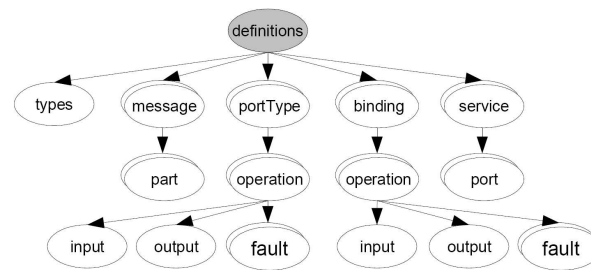


Fig. 3. WSDL structure

Figure 3 shows the WSDL document structure. It contains two sections:

- an abstract interface, describing the Web Service’s operation signatures. It includes the **operation**—organized in **portTypes**—defining the input, output, and fault **messages** composed of **parts**, which refer to a datatype, defined in the **types** section.
- a concrete implementation. It includes a **binding** section—assigning the operations to a wire format and a transport protocol—and the **ports**—defining the service’s network endpoint address.

The WSDL specification [5] either allows a variety of concrete implementations or does not make any regulations at all. This applies for example to the grammar language used for data types (XML Schema, DTD, etc.), the encoding rules (literal, SOAP encoded, etc.) or the transport binding (SOAP, HTTP POST, HTTP GET). This creates a problem in implementing compatible systems. Thus, the WS-I⁴ Basic Profile [9] recommends a number of constraints for Web Service descriptions and SOAP messages. Some important restrictions are:

- Only XML Schema is permitted for defining data types.

⁴ The *Web Services Interoperability Organisation* was founded by leading Web Service enterprises to create interoperability guides. In these guides the Web Service specifications are restricted and rendered more precisely to ease the creation of interoperable implementations.

- The only wire format is "literal" ("SOAP encoded" is not allowed, see also [1]). This means, the data types defined in the abstract section therefore become concrete types.
- Only SOAP/HTTP binding is allowed.

Additionally there exist two different Web Services styles: *RPC* and *document*. In many cases both styles result in similar wire messages, but their definitions inside a Web Service description varies in many ways. This must be taken into consideration when analysing a Web Service description. Due to space limitations, we do not discuss this problem any further.

The next sections shows how a WSDL document is analysed and compiled to an XML Schema representing exactly the messages defined by this document.

4.2 Compiling a Web Service Description

The SOAP message's structure belonging to a Web Service description is defined by informations spread all over the description document. The description must be traversed and the informations necessary for a specific service or operation must be merged into a message definition. The arrow in figure 4 shows how a Web Service description is traversed to determine the SOAP message's structure and to generate the appropriate XML Schema.

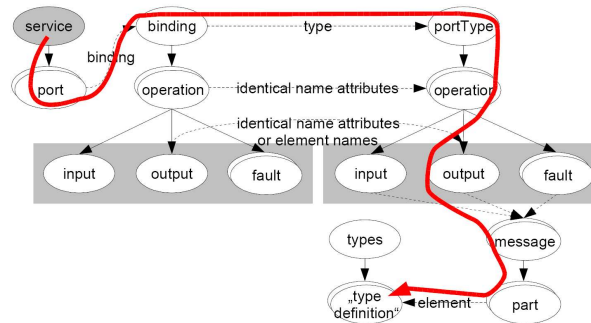


Fig. 4. Traversing a Web Service description while analysing

Figure 6 shows a simple Web Service description⁵ for a sample Web Service belonging to the SOAP message in figure 1. This example illustrates the traversing steps through a Web Service description (printed in *italics*).

The following WSDL elements are passed in this order:

⁵ The Web Service description was simplified for the sake of readability: all namespace prefixes and namespace declarations were removed; further on the declaration for the outgoing message was also omitted.

1. The Web Service contains one or more ports.
*The Web Service **AddService** contains the port **AddServicePort**.*
2. A port references a binding by the **binding**-attribute.
*The port is bound to the binding **AddServiceBinding**.*
3. A binding describes the wire format and the network protocol for the Web Service operations. All non-SOAP bindings and non-literal encodings (which do not conform to the WS-I Basic Profile) are ignored. This is also the place, where the service's style (RPC or document) is defined. Finally the binding references to a port type using the **type** attribute.
*The binding **AddServiceBinding** defines a SOAP binding using HTTP transport and literal encoding and furthermore an operation **add** with an input message using literal encoding and it links to the port type **AddServiceType**.*
4. For each operation defined in the binding section the referenced port type contains an operation with the same name and the same input, output and fault messages, which reference a message element.
*The port type **AddServiceType** contains also an operation **add**, which references the message **addSoapIn**.*
5. The message contains one or more message parts, which reference an XML Schema type or an XML Schema element (depending on the service's style) inside the types section.
*The message **addSoapIn** contains one part that links to the XML Schema element **add**.*
6. The types section defines XML data types and elements for the SOAP messages.
*The element **add** is defined as complex type containing a list of elements **x** of type integer.*

After passing a Web Service description in the way stated above the CheckWay compiler generates the appropriate XML Schemas. Figure 6 also shows the XML Schemas for the sample Web Service. The first schema defines a common SOAP message skeleton with envelope, header and body. The child of the **body** element in our simple example is just the element referenced by the message, the **add** element (as also can be seen in figure 1). Thus, the **add** element is referenced in the first schema and defined in the second schema. The definition is split into two schemas, one for each target namespace.

For RPC style the SOAP message (and so also the XML Schema for the SOAP message) is more complex and goes beyond this article's scope. For further details on creating XML Schemas from a Web Service description see [19].

To fight DoS attacks effectively—as shown in section 3.1—messages that are to be forwarded to a Web Service server must not only be valid with respect to the XML Schema that can be derived from the Web Service description, but moreover to a "hardened" XML Schema, constructed from the initial XML Schema as follows:

- Replacing **maxOccurs="unbounded"** in complex data types with an adequate number, e.g. **maxOccurs="1000"**. For most practical cases it is easy to determine an upper bound for the number of elements. With this limitation it is no longer possible to "flood" a Web Service with a endless series of elements.

- Replacing simple types without length restriction (e.g. `xsd:string`) with a corresponding data type containing a length restriction. This can be implemented by adding an XML Schema facette [8] to the simple type definition inside the `types` section of the Web Service description. Restricting simple types is easier and more natural than limiting the message’s total length (done e.g. when defining input forms or database fields).
- Removing all operations, which are not intended to be called from the internet.

The first two points restrict implicitly the total document length and thereby prevent the *Oversize Payload* attack. It has to be noted that these modifications have to be performed with respect to the concrete Web Service application. There are no universally valid values for the number of elements or the length of simple types.

Now that we have shown, how an effective XML Schema is derived from a Web Service description, we regard how XML Schema validation is implemented in our Web Service firewall.

5 The Web Service Firewall Implementation

The core of the *CheckWay Gateway* (see figure 2) is an XML validation engine, which validates the SOAP message to the appropriate schemas. If the validation is successful, the SOAP message is forwarded. SOAP messages containing an ”unlimited” number of elements do not match the (hardened) schema and are rejected. Additionally ”ultra long” simple type elements do not match the (restricted) simple type definition and are also refused.

The validator’s implementation is crucial for the gateway’s efficiency. First of all, if the gateway is vulnerable to the attacks that it is actually supposed to protect against, it is useless. Furthermore, the processing speed is—like for every network intermediary—extremely important. The gateway should not increase the total response time significantly. We developed a special XML validation engine which was designed using the following principals [16]:

- Consistent event-based XML processing
- Support for large cardinalities
- Support for all XML Schema simple types including facets

As stated before, XML parsing and validating can be very memory consuming using an improper implementation. A DOM based parser e.g. builds the complete XML document in memory. This makes it vulnerable for attacks using documents with ”unlimited” length. The gateway memory would be exhausted before the validator could even start the validating process. Thus, our validator works entirely event-based, using a SAX [15] interface. The XML document is parsed and sent event per event to the validator. The validator operates directly on these events to validate the document. There is no need, neither for the parser

nor the validator, to reconstruct the whole document in memory. In fact the validator has constant memory consumption (only depending on the schema size) and linear runtime.

The gateway can therefore easily process very large documents. If the validator finds a schema violation inside a SOAP message, the gateway has read the document only up to that particular element. The remaining document is in this case never read and can therefore not impact the gateway's function.

Theoretically, the CheckWay gateway can operate completely *on the fly*. It can forward the XML document parts that have already passed the validator. In this case, the gateway's memory usage would be completely independent from the SOAP messages' size. However a security gateway should not forward any document parts before it has stated that the whole document does not contain any malicious parts. Thus, the CheckWay stores the document until the validating process has been successfully completed. This way, only the valid document parts are stored and a Coercive Parsing attack can still not harm the gateway.

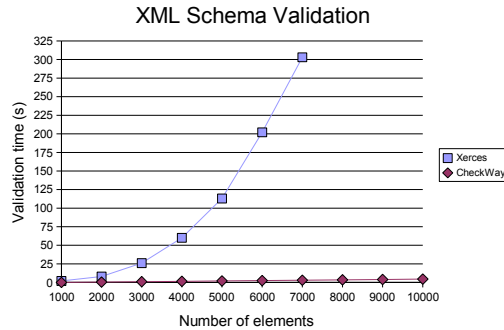


Fig. 5. Validation time

As stated earlier, the CheckWay compiler replaces `maxOccurs="unbounded"` cardinalities with a large integer. Thus the validator must be able to cope with such a schema construct, which is not self-evident. We validated documents against a simple schema with increasing `maxOccurs` value and compared our validator to Xerces⁶. Figure 5 shows the validation time for both engines. While the CheckWay validator's runtime has linear dependency on the `maxOccurs` value, the Xerces validator's runtime increases exponentially. The Xerces' time consumption, even for small values, is unacceptable for a network gateway⁷. For values greater than approx. 7500, the Xerces validator aborted throwing an out-of-memory exception.

⁶ Xerces2 Java 2.7.1

⁷ 2 seconds on a 2 GHz machine for `maxOccurs="1000"`

Together with the checking of length restricted data types created by the CheckWay compiler, the validator is able to detect the attacks pictured above.

6 Summary and Outlook

In this paper we have shown how Web Services open up new possibilities for Denial of Service attacks. We presented a solution that uses XML Schema validation to detect malicious SOAP messages. Our Web Service firewall combines a WSDL compiler to generate the necessary Schemas and an efficient XML validator to filter the potential dangerous SOAP messages.

A further kind of DoS attacks not discussed in this articles forces a server into expensive cryptographic computations. With WS-Security and WS-SecurityPolicy such attacks can also harm Web Services. We are already working on an extended Web Service firewall with security and policy support to fend off such attacks just as well.

References

- [1] Frank Cohen. Discover SOAP encoding's impact on Web service performance. *IBM developerWorks*, 2003.
- [2] Arnaud Le Hors et al. Document Object Model (DOM) Level 3 Core Specification. *W3C Recommendation*, 2004.
- [3] Bob Atkinson et al. Web Services Security (WS-Security). 2002.
- [4] David Booth et al. Web Services Architecture. *W3C Recommendation*, 2004.
- [5] Erik Christensen et al. Web Services Description Language (WSDL). *W3C Note*, 2001.
- [6] Giovanni Della-Libera et al. Web Services Security Policy Language (WS-SecurityPolicy). 2005.
- [7] H.S. Thomson et al. XML Schema Part 1: Structures Second Edition. *W3C Recommendation*, 2004.
- [8] H.S. Thomson et al. XML Schema Part 2: Datatypes Second Edition. *W3C Recommendation*, 2004.
- [9] Keith Ballinger et al. Basic Profile Version 1.1. *WS-I Organisation*, 2004.
- [10] Martin Gudgin et al. SOAP Version 1.2 Part 1: Messaging Framework. *W3C Recommendation*, 2003.
- [11] Steve Anderson et al. Web Services Secure Conversation Language (WS-SecureConversation). 2005.
- [12] Steve Anderson et al. Web Services Trust Language (WS-Trust). 2005.
- [13] Pete Lindstrom. Attacking and Defending Web Service. *A Spire Research Report*, 2004.
- [14] Brett McLaughlin. *Java and XML Data Binding*. O Reilly, 2002.
- [15] The SAX Project. Simple API for XML – SAX 2.0.1. 2002.
- [16] Florian Reuter. Forthcoming dissertation.
- [17] Günter Schäfer. Sabotageangriffe auf Kommunikationsstrukturen: Angriffstechniken und Abwehrmaßnahmen. *PIK 28*, pages 130–139, 2005.
- [18] Andre Yee. Protecting Your Web Services Deployment.
- [19] Jesper Zedlitz. Spezifikation und Implementierung eines Application Level Gateways für Web Service. *Diploma thesis*, 2004.

Sample Web Service Description:

```
<definitions targetNamespace="http://example.com/AddService">
  <types>
    <schema targetNamespace="http://example.com/AddService">
      <element name="add">
        <complexType>
          <sequence>
            <element minOccurs="1" maxOccurs="unbounded" name="x" type="int" />
          </sequence>
        </complexType>
      </element>
    </schema>
  </types>
  <message name="addSoapIn">
    <part name="parameters" element="add" />
  </message>
  <portType name="AddServiceType">
    <operation name="add">
      <input message="addSoapIn" />
    </operation>
  </portType>
  <binding name="AddServiceBinding" type="AddServiceType">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" />
    <operation name="add">
      <soap:operation soapAction="http://example.com/AddService/add" style="document" />
      <input>
        <soap:body use="literal" />
      </input>
    </operation>
  </binding>
  <service name="AddService">
    <port name="AddServicePort" binding="AddServiceBinding">
      <soap:address location="http://ws-server.local/AddService/Service1.asmx" />
    </port>
  </service>
</definitions>
```

Resulting XML Schemas:

```
<schema targetNamespace="http://schemas.xmlsoap.org/soap/envelope/">
  <element name="Envelope" type="tn:EnvelopeType" />
  <complexType name="EnvelopeType">
    <sequence>
      <element name="Body" type="tn:BodyType" />
    </sequence>
    <anyAttribute namespace="##other" />
  </complexType>
  <complexType name="BodyType">
    <choice>
      <element ref="add" />
    </choice>
    <s:anyAttribute namespace="##other" />
  </complexType>
</schema>

<schema targetNamespace="http://example.com/AddService">
  <element name="add">
    <complexType>
      <sequence>
        <element maxOccurs="1000" minOccurs="1" name="x" type="s:int" />
      </sequence>
    </complexType>
  </element>
</schema>
```

Fig. 6. Sample Web Service and generated XML Schemas