Journal "Computer Science – Research and Development", manuscript No.
(will be inserted by the editor)

Meiko Jensen · Nils Gruschka · Ralph Herkenhöner

A Survey of Attacks on Web Services

Classification and Countermeasures

Preliminary Version

Abstract Being regarded as the new paradigm for Internet communication, Web Services have introduced a large number of new standards and technologies. Though founding on decades of networking experience, Web Services are not more resistant to security attacks than other open network systems. Quite the opposite is true: Web Services are exposed to attacks well-known from common Internet protocols and additionally to new kinds of attacks targeting Web Services in particular. Along with their severe impact, most of these attacks can be performed with minimum effort from the attacker's side.

This article gives a survey of vulnerabilities in the context of Web Services. As a proof of the practical relevance of the threats, exemplary attacks on widespread Web Service implementations were performed. Further, general countermeasures for prevention and mitigation of such attacks are discussed.

Keywords Web Services · Security · Attacks · Denial of Service · Flooding Attacks · XML · WS-Security

CR Subject Classification C.2 · C.4 · H.3.5 · K.6.5

M. Jensen
Horst Görtz Institute for IT-Security,
Ruhr University Bochum, Germany
Tel.: ++49-234-32-26796
Fax: ++49-234-32-14347
E-Mail: Meiko.Jensen@ruhr-uni-bochum.de

N. Gruschka
NEC Europe Ltd.
NEC Laboratories Europe, IT Research Division
St. Augustin, Germany
Tel.: ++49-2241-9252-30
Fax: ++49-2241-9252-99
E-Mail: gruschka@it.neclab.eu

R. Herkenhöner
Institute for IT-Security and Security Law
University Passau, Germany
Tel.: ++49-851-509-3026
Fax: ++49-851-509-3052
E-Mail: ralph.herkenhoener@uni-passau.de

This work was done while the authors were at the Department for Computer Science, University of Kiel, Germany

1 Introduction

Web Services and *Service-Oriented Architectures* (SOAs) are often considered to be among the most important technological innovations of the last decade. Nevertheless, the benefits of these new approaches stand against some serious flaws these new technologies bring along. The most severe issues concern Web Service security [19].

The typical requirements for a secure system are *integrity*, *confidentiality* and *availability*. Any action targeting at violation of one of these properties is called an *attack*, the possibility for an attack is called a *vulnerability*.

This article presents a list of security issues in the domain of Web Services. The list does not claim to be complete, it merely is a selection of the most impressive attacks we examined during our research. As this research focused on availability, most of the attacks belong to the category of *Denial-of-Service* (DoS) attacks [22].

The severity of DoS attacks can be seen in daily news, for example the Distributed Denial-of-Service (DDoS) attacks on Estonian governmental and commercial web sites in April/May 2007 [25]. These attacks were performed by botnets using network layer flooding techniques. As we will show in this article, DoS attacks on Web Services can be conducted with much less resource effort than against non-Web-Service systems.

The attacks cover a wide range of aspects. Starting with attacks on single Web Services without security measures, we further present attacks on *WS-Security*-enabled Web Services, and finally describe attacks on Web Services used in Web Service compositions. Although the latter are applicable for all types of Web Service compositions, we have chosen *WS-BPEL* (or BPEL for short) for attack demonstration, as it tends to become the leading Web Service composition standard.

The remainder of this article is organized as follows. In the next section, the basic concepts and terminologies of Web Service security and BPEL are introduced. Section 3 lists vulnerabilities and attacks on Web Services. Section 4 then discusses general countermeasure concepts, and Section 5 provides an attack classification scheme. Finally, in Section 6 we conclude about the work presented in this article.

2 Fundamentals

2.1 WS-Security

The most important specification addressing the security needs of Web Services is WS-Security [21]. It collaborates with the SOAP specifications, providing integrity, confidentiality and authentication for Web Services. WS-Security defines a SOAP header block—the so-called *security header*—that carries the WS-Security extensions. Additionally, it defines how existing XML security standards like *XML Signature* [2] and *XML Encryption* [13] are applied to SOAP messages.

XML Signature allows XML fragments to be digitally signed to ensure integrity or to proof authenticity. The result of the signing operation—i.e. the encrypted digest—is placed in a `Signature` element, which again is added to the security header.

XML Encryption allows XML fragments to be encrypted to ensure data confidentiality. The encrypted fragment is replaced by an `EncryptedData` element containing the ciphertext of the encrypted fragment as content.

Further, XML Encryption defines an `EncryptedKey` element for key transportation purposes. The default application for an encrypted key is a *hybrid encryption*: an XML fragment is encrypted with a randomly generated symmetric key, which itself is encrypted using the public key of the message recipient. In SOAP messages, the `EncryptedKey` element—if present—must appear inside the security header.

In addition to encryption and signatures, WS-Security defines security tokens suitable for transportation of digital identities, e.g. `UsernameToken` or X.509 certificates.

An important characteristic of the mechanisms used in WS-Security is their high flexibility. They are applicable to arbitrary parts of the SOAP message, leaving all other parts unattended. As a consequence, Web Service servers and clients must negotiate a security policy defining the WS-Security elements to be used.

WS-SecurityPolicy [17] provides an XML syntax for declaring such security policies. In extension to the Web Service description, a server may use a WS-SecurityPolicy document for declaring its security needs. WS-SecurityPolicy allows to specify the parts of a SOAP message that shall be encrypted or signed, the algorithms to use and the required security tokens.

2.2 BPEL

The *Business Process Execution Language* [1] is one of the favorite candidates to become the predominant *Web Service composition* standard. Each BPEL document describes the workflow of a so-called *BPEL process*. Such a process consists of *activities*, representing instructions to be executed by a BPEL runtime environment—the *BPEL engine*. These activities can be categorized into *communication activities* representing incoming or outgoing Web Service calls, *structure activities* for execution order description, and other *basic activities* for additional tasks, such as process variable access, temporal constraints in workflow execution or fault handling. At runtime, each deployed BPEL process may have multiple *process instances*, which are concurrent execution contexts of the same process.

One key feature of BPEL-based Web Service composition is the ability to use *asynchronous communication*. A regular Web Service call consists of a request message, directly answered by a reply message. The requester must keep the connection to the server until the reply message arrives. Using a special language construct, BPEL enables asynchronous behavior, allowing the requester to disconnect after sending its request. In this case, the reply message is delivered via a new connection initiated by the Web Service server, e.g. by invoking a Web Service on the original requester. This communication pattern is useful for long-running tasks that cannot be completed within timeout limits of a single Web Service call.

The specification in use for specifying the callback destination is WS-Addressing [11], allowing the requester to specify an abstract *endpoint reference* within its request message, containing all information necessary for the BPEL engine to invoke the Web Service on the requester.

A further task a BPEL engine has to perform is message correlation. As a BPEL engine may run several instances of one BPEL process at the same time, it becomes necessary to use designated message data fields to identify the target process instance for an incoming Web Service message. These are called *correlation sets* in the context of BPEL.

3 Attacks

In this section we present a list of attacks on Web Services. For each attack an abstract attack methodology and impact is given, demonstrated by a concrete attack execution where appropriate. Additionally, countermeasures against the particular attacks are discussed.

3.1 Oversize Payload

One important category of Denial-of-Service attacks is called *Resource Exhaustion* [24]. Such attacks target at eliminating a service's availability by exhausting the resources of the service's host system, like memory, processing resources or network bandwidth. One "classic" way to perform such a Resource Exhaustion attack is to query a service using a very large request message. This is called an *Oversize Payload* attack [19].

Against Web Services, an Oversize Payload attack is quite easy to perform, due to the high memory consumption of XML processing. The total memory usage caused by processing one SOAP message is much higher than just the message size. This is due to the fact that most Web Service frameworks implement a tree-based XML processing model like the *Document Order Model* (DOM [12]). Using this model, an XML document—like a SOAP message—is completely read, parsed and transformed into an in-memory object representation, which occupies much more memory space than the original XML document. For common Web Service frameworks, we observed a raise in memory consumption of factor 2 to 30.

Example: An Axis Web Service was attacked using a large SOAP message document, which consisted of a long list of elements considered as parameter values of the Web Service operation[1]:

```
<Envelope>
  <Body>
    <getArrayLength>
      <item>x</item>
      <item>x</item>
      <item>x</item>
      ...
    </getArrayLength>
  </Body>
</Envelope>
```

The SOAP message had a total size of approx. 1.8 MB. The message processing induced a full CPU load for more than one minute and an additional memory usage of more than 50 MB. Enlarging the message to approx. 1.9 MB even resulted in an out-of-memory exception.

An obvious countermeasure against Oversize Payload attacks consists in restriction of the total buffer size (in bytes) for incoming SOAP messages. In this case, it is sufficient to check the actual message size and reject any message exceeding the predefined limit. This method is used by the .NET 2.0 framework, which discards all SOAP messages larger than 4 MB (in the default configuration). While this countermeasure is very simple to implement, it is not suitable for Web Service messages.

A more appropriate approach uses restrictions on the XML infoset. This can be realized by modifying the XML schema inside the Web Service description and validating incoming SOAP message to this schema [7]. Details of this approach can be found in section 4.

[1] In all sample documents, namespaces were omitted for simplification reasons.

3.2 Coercive Parsing

One of the first steps in processing a Web Service request is parsing the SOAP message and transforming the content to make it accessible for the application behind the Web Service. Especially when using namespaces, XML can become verbose and complex in parsing, compared to other message encodings. Thus, the XML parsing process allows other possibilities for a special kind of Denial-of-Service attacks, which is called *Coercive Parsing* attacks [19].

Example: The following attack was performed targeting an Axis2 Web Service. The attack used a continuous sequence of opening tags:

```
<x>
  <x>
    <x>
      ...
```

The attack resulted in a CPU usage of 100% on the target system. The service's availability was massively reduced, and the incoming message was finally received with a constant rate of 150 byte/s. Thus, the attack would perform well even if the attacker has a very low bandwidth connection. The Web Service server did not abort the connection, thus this attack could apparently be continued infinitely. In our experiment, we stopped the attack after one hour.

Typical Coercive Parsing attacks targeting at resource exhaustion use a large number of namespace declarations, oversized prefix names or namespace URIs or very deeply nested XML structures. These types of attacks require different countermeasures.

An attack that is based on complex or deeply nested XML documents (like the one in the example above) can be fended by using schema validation (compare section 4).

Attacks misusing namespace declarations are harder to prevent. As the XML specification does neither limit the number of namespace declarations per XML element nor the length of the namespace URIs, any restriction on the number or length of namespace declarations would be arbitrary and could lead to unpredictable rejection of messages.

3.3 SOAPAction Spoofing

The actual Web Service operation addressed by a SOAP request is identified by the first child element of the SOAP body element. Additionally, the optional HTTP header field "SOAPAction" can be used for operation identification. Although this value only represents a hint to the actual operation, the SOAPAction field value is often used as the only qualifier for the requested operation. This is based on the bogus optimization that evaluating the HTTP header does not require any XML processing.

This twofold operation identification enables two classes of attacks. The first one is executed by a man-in-the-middle attacker and tries to invoke an operation different from the one specified inside the SOAP body. It is based on modification of the HTTP header.

Example: The following attack was performed targeting a .NET Web Service. The deployed service provided two operations: `op1(string s)` and `op2(int x)`—with the respective SOAPAction and message element also named op$n$. The following message (including the HTTP header) was sent to the service:

```
POST /Service.asmx HTTP/1.1
...
SOAPAction: "op2"

<Envelope>
  <Body>
    <op1>
      <s>Hello</s>
   </op1>
  </Body>
</Envelope>
```

The method call that was triggered by this message was: `op2(0)`. This shows that the operation is selected solely by the SOAPAction value from the HTTP header. Even worse, the “wrong” operation was executed despite of incompatible parameter names and types.

The example shows how modifications of the HTTP header can invoke methods that were not intended by the SOAP message creator. As the HTTP header is not secured by WS-Security and is newly created at every SOAP intermediary, it can easily be modified.

The second class of SOAPAction spoofing attacks is executed by the Web Service client and tries to bypass an HTTP gateway.

Example: The following attack was performed targeting an Axis2 Web Service. The deployed service provided two operations: `hidden` and `visible`—with the respective SOAPAction and message element equally named. The following message (including the HTTP header) was sent to the service:

```
POST /axis2/testService HTTP/1.1
...
SOAPAction: "visible"

<Envelope>
  <Body>
    <hidden />
  </Body>
</Envelope>
```

The Axis2 server actually ignored the SOAPAction value and invoked the `hidden` operation instead. If an HTTP border gateway—which of course operates on the HTTP header only—is configured to reject `hidden` and accept `visible` accesses, this attack allows calling `hidden` anyway.

A countermeasure to SOAPAction Spoofing attacks would be to determine the operation by the SOAP body content. Additionally, the operations determined by the HTTP header and by the SOAP body must be compared and any difference should be regarded as threat and result in rejecting the Web Service request.

3.4 XML Injection

An *XML Injection* attack tries to modify the XML structure of a SOAP message (or any other XML document) by inserting content—e.g. operation parameters—containing XML tags. Such attacks are possible if the special characters ”<” and ”>” are not escaped appropriately. At the Web Service server side, this content is regarded as part of the SOAP message structure and can lead to undesired effects.

Example: The following attack was executed against a .NET Web Service. The deployed service method has two parameters `a` and `b`, both of type `xsd:int`. This service was invoked using the following SOAP message:

```
<Envelope>
  <Body>
    <HelloWorld>
      <a> <b>1</b> </a>

     <b> 2 </b>
   </HelloWorld>
  </Body>
</Envelope>
```

Such a message could result from an XML Injection attack: `<b>1</b>` was inserted as parameter content without escaping ”<” and ”>”. As the SOAP message obviously violates the Web Service schema, it should be rejected. But in fact, not only that the message was accepted by .NET, the resulting parameter values inside the service application for this request were: $\mathtt{a} = 1$, $\mathtt{b} = 0$. Thus, the attacker was able to modify the value of `b` just by modifying the content of `a`. It is easy to imagine a scenario in which this can lead to unintended service behavior, e.g. access to restricted data.

An important step in detecting such attacks is a strict schema validation on the SOAP message, including data type validation as possible (see section 4). This would have rejected the message from the example attack.

3.5 WSDL Scanning

The WSDL advertises a service's operations including parameters, data types and network bindings. Usually, some of these operations should be accessed from the local network only (here called *internal* operations), while other operations are intended to be offered to the outer network (here called *external* operations).

If the Web Service is created using common Web Service framework tools, the (only) generated WSDL contains all operations. In this case, an external client gains knowledge of the internal operations and can invoke them.

The first step in avoiding such accesses is providing a separate WSDL to external clients that contains the external operations only. However, as the Web Service endpoint is still externally accessible (for invoking the external operations), an attacker can try to guess the omitted operations and call them. This attack is called *WSDL Scanning*.

For example, an internet shop system needs methods for placing an order for customers (`sendOrder`) and for administrating the orders (`adminOrders`). Of course, the latter one is intended to be called from within the shop's intranet only. If both `sendOrder` and `adminOrders` operations are offered by one Web Service, an attacker with the knowledge of `sendOrder` can easily find the administration method also. Other examples for vulnerable internal operations are legacy and debug methods.

One countermeasure to this attack is deploying the internal and external operations to separate Web Services, preferable even on different servers. If this is not applicable, invocations of internal operations must be controlled and rejected if originated from an external client. This is a typical task for a border gateway. Unfortunately, external and internal request messages have the same destination IP address, TCP port and even HTTP URL. Thus, packet filters and HTTP firewalls can not decide whether the Web Service operation is allowed or not. Therefore, a Web-Service-aware XML firewall is required which is configured with the externally visible operations.

3.6 Metadata Spoofing

A Web Service client retrieves all information regarding a Web Service invocation (i.e. message format, network location, security requirements etc.) from the metadata documents provided by the Web Service server. Currently, this metadata usually is distributed using communication protocols like HTTP or mail. These circumstances open new attack possibilities aiming at *spoofing* these metadata. The most relevant attacks in this category are *WSDL Spoofing* and *Security Policy Spoofing*.

Supposably most promising for WSDL Spoofing is the modification of the network endpoints and the references to security policies. A modified endpoint enables the attacker to easily establish a man-in-the-middle attack for eavesdropping or data modification. If additionally a spoofed security policy with lower or no security requirements is used, such attacks are possible despite the use of WS-Security.

To avoid Metadata Spoofing, all metadata documents must be carefully checked for authenticity. However, the mechanisms for securing metadata documents are not standardized—in contrast to methods for securing SOAP messages. Additionally, a prior establishment of trust relationships is required, which is not always possible or intended.

3.7 Attack Obfuscation

Using WS-Security on Web Services introduces new problems concerning service availability. By providing confidentiality to sensible data, XML Encryption can mask message content from being inspected. As this encrypted content can contain an intended attack—like Oversize Payload, Coercive Parsing or XML Injection—encryption can be used to conceal attacks.

Most problematical is that this kind of attacks is hard to detect. To analyze the message structure—e.g. for schema validation—decryption is necessary. There are two possibilities on how a targeted system may be effected. If decryption is done after message validation, the malicious message content may pass the message validation. If decryption is done before message validation, the system may tie up during message decryption because of the XML and cryptographic processing. Thus, even if a system is able to counter the unencrypted attack, obfuscated attacks may affect a target system anyway.

Example: During tests with Axis2 a weak spot for obfuscated Oversize Payload attacks was revealed. In this scenario a single SOAP message[2] was sent to the server. Sending a message containing an encrypted and signed body with a size of 1 MB caused a full CPU load for 23 seconds and resulted in an out-of-memory exception, while the Java runtime environment additionally consumed approx. 90 MB of system memory. In comparison, unencrypted messages with message sizes of 20 MB and more were processed by the Axis2 server within a processing time of beneath one second.

To counter obfuscated attacks, a good strategy is performing message validation on decrypted content. The best effort uses a stepwise decryption and validation. This can help reducing memory consumption and enables an early detection of malicious message content.

3.8 Oversized Cryptography

Another problem introduced by WS-Security is the flexible usability of security elements: encryption may be

[2] The message was generated using also the Axis2 framework.

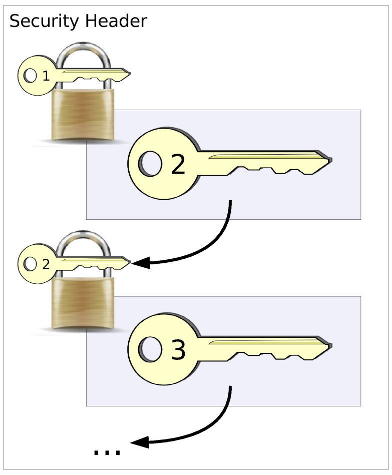

Abb. 1 Example for an encrypted key chain (schematically)

used almost anywhere within a SOAP message, and the flexible layout of the security header allows no strict schema validation. The various possibilities for using security elements limits a schema validation to check each single element, but neither order nor occurrence checks for multiple elements are possible. This flexibility can be misused for attacks.

A self-evident attack relies on an oversized security header. If the target system processes or buffers the whole security header, the target system may be effected the same way as from an Oversize Payload attack (see section 3.1).

A more complex attack with an oversized security header uses chained encrypted keys. In this chain, each encrypted key is used to encrypt the next key (see figure 1). Thus, the target system is forced to decrypt every encrypted key, as each key is needed for decryption of the next one. This may effect the target system in two ways. First, the target system must buffer every key, as it is unknown before the end of message processing, whether an encrypted key is used for other encrypted content. This leads to high memory consumption. Second, the decryption operations needs processing resources. Especially asymmetric algorithms, which are typically used for key transport, are highly CPU consuming.

A similar attack uses a large number of nested encrypted blocks within a SOAP message (like a Russian matryoshka doll). The target system must decrypt all the nested blocks in order to process the inner content. This induces a high CPU load due to the large number of cryptographic operations. Additionally, if the decrypted content is buffered before further processing, the memory consumption is a multiple of the original message size.

As a countermeasure, the usage of WS-Security elements must be restricted, and messages exceeding these limits must be rejected. In contrast to the *Oversized Payload* and the *Attack Obfuscation* attack, schema based restrictions are only partly effective: the security header schema allows any kind and amount of security tokens, and encrypted blocks are allowed nearly everywhere within the SOAP message.

A better approach is accepting only the security elements explicitly required by the security policy. This is called *Strict WS-SecurityPolicy Enforcement* and is futher explained in section 4.

3.9 BPEL State Deviation

As BPEL processes need to be called by external communication partners, a BPEL engine provides Web Service endpoints accepting every possible incoming message. Due to the fact that one BPEL process may have many *process instances* running concurrently, these communication endpoints are open for incoming connections at any time. Thus, a malicious Web Service client might attack these open Web Service endpoints using messages that are correct regarding their message structure, but that are not properly correlated to any existing process instance. These *correlation-invalid* messages will be discarded within the BPEL engine, but they cause a huge amount of redundant work. Each message must be read and processed completely, searching all existing process instances for a match, before the message may safely be discarded. Thus, the computational resources of the BPEL engine get exhausted by processing such invalid messages.

Example: The following attack was executed against a BPEL engine running one BPEL process. The process contained amongst other activities a sequence of two receive activities `first` and `second`, with only `first` initiating a new process instance. Additionally, the process defines a number of correlation properties for process instance identification. The attack used SOAP messages invoking operation `second` and containing correlation properties that did not match to any running process instance. The BPEL engine was attacked by a sequence of 1000 messages, summing up to a total payload of 0.5 MB. The attack messages were correctly discarded by the BPEL engine but resulted in an additional memory consumption of 350 MB and a full CPU load for more than 2 hours.

A second subtype of this attack uses correct correlation properties, but targets a `receive` activity that is not enabled in the actual state of the instance's process execution. These messages are not correlation-invalid but *state-invalid*. Their impact instead is the same: resource exhaustion on the BPEL engine's processing resources, leading to a reduced quality of service or even a loss of availability.

To fend state deviation attacks, it is necessary to identify and reject correlation-invalid and state-invalid messages, using as few computational resources as possible. Note that the identification of state deviation attack messages differs widely for these two message types. A firewall approach fending both attack types was described in [10] and [15].

3.10 Instantiation Flooding

Every BPEL-based workflow definition contains at least one communication activity that creates a new process instance each time a message arrives. Such a process instance immediately starts its execution according to the instructions given in the process description. The execution will be paused each time a receive activity is reached, continuing after reception of the expected message. By reaching a termination point, execution is stopped and the process instance is destroyed. Note that all execution activities except `receive` activities are completely driven by the BPEL engine alone. Just in case of `receive`, an external message triggers the next executions.

Keeping this in mind, imagine an attack that continuously calls the instantiating activity's endpoint. For each incoming SOAP message, the BPEL engine will create a new process instance and start its execution, running each of these until they reach either a receive activity or a termination point. As a result, the BPEL engine will get into heavy load for message parsing, process instantiation and activity execution, which will decrease or even nullify the availability of the BPEL engine.

When examining instantiation flooding in the context of BPEL, there are some behavioral distinctions to make on the attack's impact.

First, you have to realize that the BPEL engine itself is not the only target reached by this attack. Each newly created process instance is executed just like a valid one, including all its outgoing Web Service requests to external communication partners. Thus, these communication partners will undergo a raise in requests initiated by the BPEL engine as well (see next section).

Further, note that the processing of a single attack message and thus the resource exhaustion impact is determined by one of the following circumstances.

– If the process under attack does not contain any receive activity (beside the initial one), all executions will stop when reaching a termination point, destroying the initially created process instance. Thus, the BPEL engine will undergo the "usual" traffic for each attack message.
– If there exists a `receive` (or `pick`) activity on the execution path chosen for the incoming attack message, all created process instances will run up to that receive activity. Here, according to the BPEL specification [1], all but the first request will cause a BPEL execution fault, as there already exists a receiving process instance with the same correlation data. Thus, all but the first process instance will execute the fault handler (if existing in the process description), causing a rollback on all activities that have already been executed previously. This will double the resource load for both BPEL engine and previously requested communication peers that are included in rollback procedures.
– If there exists a `receive` (or `pick`) activity on the execution path, and the attack uses unique correlation data for each attack message (such as including a counter's value in one of the correlation data fields), none of the messages will cause a fault as stated above. Instead, all attack messages will cause creation of new process instances that all will run up to the first `receive` activity and wait there forever (or—in case of `pick`—as long as the timeout case specifies). As a result, the BPEL engine will need a huge amount of persistent storage for keeping those process instances that never will complete. Further, the task of identifying the correct process instance for each message arriving at such an overfull communication endpoint will become really hard, as the message's correlation data must be compared to that of each process instance waiting. Due to the huge number of such waiting process instances, this task will exhaust resources a lot more than before the attack.

The overall impact of both attack types depends on the structure of the BPEL process, but it potentially is a multiple of the load necessary to perform the attack. Further, it may reach not just the BPEL engine, but some of its communication partners as well (see next section).

Fending such flooding attacks can only be achieved by identification and rejection of semantically invalid requests (attack messages). This is a non-trivial task, as the decision on whether an incoming message is semantically invalid can only be made *after* it has been processed and identified. But even at this stage it is hardly determinable whether a request was valid or malicious, as the semantics of a process usually are not included in the process description.

3.11 Indirect Flooding

Using the same methodology as presented in the previous section, the attack target of the indirect flooding attack differs. The idea of this attack is to use the BPEL engine as intermediate for an attack on a target system "behind" the BPEL engine. Imagine an architecture as shown in figure 2, and think of a BPEL process that repeatedly calls a Web Service provided by the attack target system, for example creating customer accounts with several details.

By flooding the process within the BPEL engine with instantiating attack messages (as shown in the previous section), the BPEL engine will undergo a heavy load itself, but it merely will cause an equally heavy load on the target system. Thus, if the target system is not as powerful as the BPEL engine, it will loose availability, finally resulting in a Denial-of-Service.

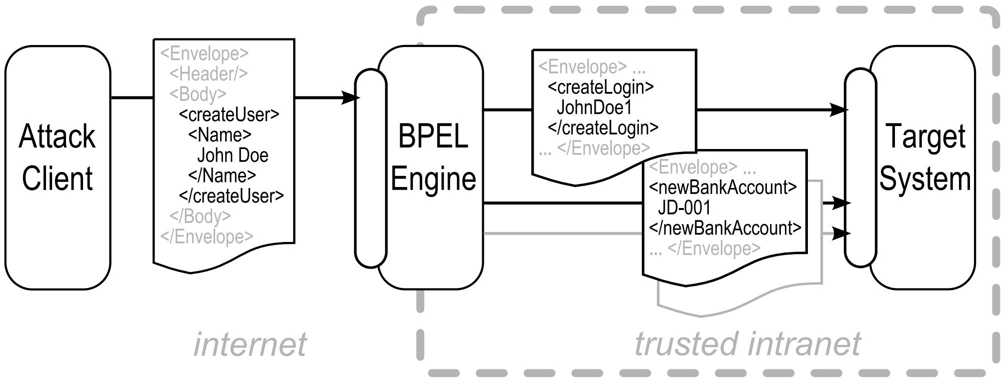

Abb. 2 Architecture for *Indirect Flooding Attack*

Using this attack method, the attacker bypasses any firewall on his direct link to the target system. Even if the target system is not connected to the outside world at all and only communicates with the BPEL engine, it stays attackable. Note that this attack method can not be fended using WS-Security or similar approaches, as the connection between BPEL engine and target system is used in a completely valid and trustful way.

Again, fending such attacks needs identification and rejection of attack messages. The complication raised here is that the responsibility for attack prevention is at the BPEL engine, but the impact is on the target system. Thinking of a scenario where BPEL engine and target system communicate over inter-corporate boundaries, this task may become a political rather than a technical problem. Further, as the workflow may spread over multiple systems hosted by multiple companies, an attack may propagate throughout the system, making it difficult to identify its real entry point.

3.12 WS-Addressing Spoofing

The use of WS-Addressing for asynchronous Web Service calls raises a lot of attack possibilities, which all have in common that they use modified callback endpoint references. The most simple approach is to use an arbitrary invalid endpoint URL as callback endpoint reference. As a result, the BPEL engine will perform the execution of the process involved, then try to callback the initiator. This will result either in a direct error (refused connection, HTTP server error or SOAP fault of any kind) or in a timeout, depending on the endpoint URL the reference denotes. Thus, the BPEL engine will raise an execution fault and call matching fault handlers and compensation handlers. All in all, the BPEL engine will execute the full process and then perform a complete rollback. Used as a flooding attack, this will cause heavy load on the BPEL engine. Compared to usual flooding attacks presented above, the workload produced by each attack message is maximized, as—in most processes—the fault will be thrown within the last communication activity of the process.

The core countermeasure against any kind of WS-Addressing spoofing is verification of the caller's endpoint URL, ideally at the beginning of a process execution. This would enable early message rejection, preventing the BPEL engine from unnecessary workload.

3.13 Middleware Hijacking

This attack uses WS-Addressing spoofing again, but it points the attacker's endpoint URL to an existing target system, running a real service at the URL specified (see figure 3). As a result, the Web Service server (e.g. a BPEL engine) will repeatedly try to "answer" the attacker's requests using this specified URL. Thus, the service under attack receives a huge amount of requests containing SOAPFaults or invalid SOAP messages (or even worse: valid ones).

As Web Service servers are usually driven by powerful server machines, it is possible that the target system will suffer a Denial-of-Service before the hijacked server does (see section 3.12). Thus, the attacker uses the power of the server host system to tear down the target system.

As an example, the Axis2 Web Service framework today is shipped with WS-Addressing module enabled by default. Thus, any Web Service driven by Axis2 potentially is vulnerable to become hijacked using WS-Addressing Spoofing. Since SOAPFault messages are always delivered to the address specified in the `<FaultTo>` SOAP header, it does not even need a valid service execution at the server to play the trick; a faulty message with appropriate `<FaultTo>` address is sufficient.

Note that—just like with the indirect flooding attack (see section 3.11)—it is possible to use this technique to attack any system behind an internet firewall. Unlike the general indirect flooding attack, the use of WS-Addressing even enables the attacker to select the target system of the attack himself.

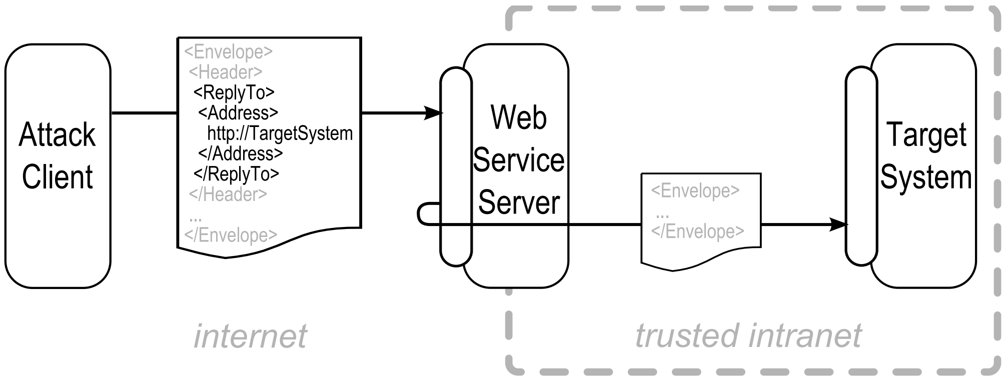

Abb. 3 Architecture for *Middleware Hijacking Attack*

4 General Countermeasure Approaches

Attacks on Web Services—as on any other system—rely on a large number of different vulnerabilities. Therefore, countermeasures against attacks are also very wide-ranging. Nevertheless, there exist several general defense mechanisms.

4.1 Schema Validation

Schema validation can be used against attacks, which use messages that are not conform to the Web Service description. Such attacks are called *deviation from protocol message syntax* [18]. By validating incoming messages to the XML schema generated from the WSDL, the attack can be detected—like shown in section 3.2 and 3.4.

Nevertheless, in current Web Service frameworks schema validation is not used or not activated by default. This is mainly due to performance reasons, as schema validation is expensive regarding CPU load and memory consumption.

Schema validation is also effective against some other attacks on Web Service applications, like SQL Injection or Parameter Tampering [19], which also use non-valid messages[3].

Additionally, schema validation can be used as a foundation for other countermeasures. One important example is restricting the XML infoset to limit the memory needed for processing the message—like discussed in section 3.1. This is what we call *Schema Hardening.*

4.2 Schema Hardening

The general idea is to analyze a schema—e.g. from a Web Service description—for constructs allowing unbounded large or complex XML trees. These constructs are modified to fulfill finite boundaries.

For example, if the Web Service description defines an unbounded list of elements[4], the list is converted into a list with limited number of elements. Inside the XML schema, the entry `<element maxOccurs="unbounded">` is replaced by `<element maxOccurs="`n`">`, where n is a finite number. For most services such a limit is easy to define. An advantage of this restriction—compared to a limit of the network buffer size—is that this limit can be included in the service's "official" Web Service description and thus becomes visible to clients in advance.

A second application of schema hardening could be removal of non-public operations from the schema inside the Web Service description (see section 3.5).

There are a number of further possibilities for hardening the Web Service description—and thus the XML schema generated. Details can be found in [7]. The same article also discusses problems raised by processing schemas containing large "maxOccurs" values.

4.3 Strict WS-SecurityPolicy Enforcement

A WS-SecurityPolicy policy defines a minimum set of security tokens that have to be included within a SOAP message to fulfill the policy. The specification does not provide a possibility for declaring their maximum usage. So—as discussed before—an attacker may add an unbounded number of additional tokens, engaging the targeted system in costly cryptographic computations and forcing high memory consumption.

To avoid this, a good strategy is to consider the requirements from the WS-SecurityPolicy document not only as a minimum requirement, but also as a maximum requirement. This means, a SOAP message must contain exactly the security tokens specified by the security policy—not less, not more.

As pointed out in [6], this limitation does not restrict the functionality, but enables the detection of attacks

[3] As these attacks are not Web Service specific, but can affect nearly every remote-accessable application, they are not discussed within this article.

[4] Such a construct is very common in practice, as Web Service framework generate it automatically for every array parameter of a Web Service method.

using oversized cryptography and can help to mitigate their effects.

4.4 Event-based SOAP message processing

The effectiveness of the countermeasures presented above highly depends on their implementation. Checking a SOAP message for conformance to the message schema and the security policy requires XML and WS-Security processing. These operations must be implemented very resource-economically, otherwise the protection system would be vulnerable to similar attacks as the Web Service itself.

Attacks using large SOAP messages make tree-based implementations like DOM unsuitable for a protection system. Such implementations require that the message must be completely read from the network and built into a document tree before the SOAP message can be further processed. Thus, before the inspection has started, a large amount of memory has already been consumed. Some tree-based implementations construct only parts of the document tree, which slightly reduces memory consumption, but does not eliminate the fundamental problem of tree-based approaches; every XML document must be completely read and stored [23].

A protection system should use an event-based XML processing model like SAX [26]. The main advantage of event-based XML processing is the possibility to detect invalid message content and abort futher processing. This way, memory consumption and CPU usage can be minimized.

The results of the example attack described in section 3.7 demonstrate the fact that Axis2 uses a stream-based message processing model (called AXIOM [14]), but Rampart—the Axis2 WS-Security component—does not [4].

While schema validation is performed in an event-based manner by a number of current implementations (e.g. Xerces, .NET), WS-Security is usually still processed using XML trees (e.g. Apache Rampart). WS-Security-enabled messages include a number of references between the WS-Security tokens, and therefore event-based evaluation is hard to realize. However, assuming some minor restrictions, it is possible to perform event-based WS-Security validation [8].

Further, in [5], Gruschka shows methods for processing and validating Web Service messages in a complete event-based manner. He also proves that the combination of event-based processing and strict protocol validation fends Denial-of-Service attacks.

4.5 WS-Security

A common misunderstanding about WS-Security is that its usage automatically ensures full security for Web Services. As shown before, WS-Security defines mechanisms for enabling integrity and confidentiality for Web Service messages. However, if the corresponding WS-SecurityPolicy is not defined correctly, attacks on integrity and confidentiality are possible using so-called *XML rewriting* attacks [3; 20]. More important in the context of this article, WS-Security does not define any direct countermeasures against attacks like Denial-of-Service.

A well-known protective mechanism for service availability is access control. Access control restricts access to the service to trusted users, which are supposed to be less "dangerous" regarding attacks. Additionally, access control enables accountability, allowing to exclude and prosecute the attacker. WS-Security defines security tokens for authentication, which can be used for access control systems.

Of course, access control can not fully eliminate the threat of attacks. First of all, even trusted communication partners can—intentionally or unintentionally—execute attacks.

Further, due to the fact that authentication needs a key infrastructure, it is not applicable in B2C relationships, as there is no wide-spread key infrastructure among private users.

Finally, the usage of WS-Security itself enables new kinds of DoS attacks, as seen in sections 3.7–3.8. Thus, authentication for Web Services must be performed in consideration of such attacks. In [9] e.g. a Denial-of-Service-robust authentication scheme for Web Service is presented.

To resume, WS-Security is one of the important building blocks for fending attacks but has to be applied carefully and is—unlike often considered—not a magic bullet against network threats.

5 Classification

In an effort to categorize and systemize these numerous attacks, we took a closer look at their specific impacts. Table 1 shows a classification of the attacks described here, based on the following parameters.

Category: Describes the security property that is violated by the attack. Possible values are confidentiality (**C**), data integrity (**I**), avaliability/Denial-of-Service (**A**) or access control issues (**AC**).

Level: This value indicates whether the attack resides on messaging layer (**M**) or on process layer (**P**) as defined in [27].

Spreading: Attacks can be application-specific (**A**), targeting a specific Web Service framework only, or they can be due to a conceptional (**C**) flaw of the underlying protocol specifications.

Size: Some attacks target single Web Services, others involve several communication partners. The Size value gives the usual or minimal number of involved systems—apart from the attacker.

Attack	Category	Level	Spreading	Size	Deviation	Dependencies	Fendability	Amplification
Oversize Payload	A	M	C	1	[S]	none	Schema validation (m), Schema hardening (m-f)	28
Coercive Parsing	A	M	C	1	[S]	none	Schema validation (m), Schema hardening (m-f)	
SOAPAction Spoofing	AC	M	A	1	S	missing comparison of operation and SOAPAction	match values (f)	
XML Injection	I	M	A	2+	A,[S]	faulty processing of client-side input data	Schema validation (m), client-side message validation (f)	
WSDL Scanning	AC,C	M	A	1	A	existence of secret operations in public WSDL	WSDL reduction (f)	
Metadata Spoofing	all	M	C	1+	A	ability to access/modify metadata documents	authenticity check of metadata documents (f)	
Attack Obfuscation	A	M	C	1	A,[S]	WS-Security processing enabled	none	90
Oversized Cryptography	A	M	C	1	[S]	WS-Security processing enabled	strict security policy enforcement (m-f)	
BPEL State Deviation	A	P	A	1	O	BPEL process with more than one inbound endpoint	stateful firewalling (m)	700
Instantiation Flooding	A	P	C	1	A,[O]	none (knowledge of correlation sets fortifies impact)	efficient correlation set matching (m)	
Indirect Flooding	A,AC	P	A	2+	A,[O]	appropriate BPEL process	none	
WS-Addressing Spoofing	A,C	M	C	2	A	WS-Addressing processing enabled	address validity / access authorization check (m-f)	
Middleware Hijacking	A,AC	P	C	2+	A,[O]	asynchronous BPEL process	address validity / access authorization check (m-f)	

Tabelle 1 Attack Classification

Deviation: Describes whether the attack generally uses syntactical (**S**), sequential (**O**), or semantical/application-specific (**A**) *protocol deviation* techniques. A [·] indicates potential, but not necessary deviation.

Dependencies: This parameter indicates how far an attack relies on prerequisites at the targeted Web Service server, e.g. the existence of a specific operation or a necessary flaw in the Web Service description.

Fendability: A measure on how effective potential countermeasures can be in terms of mitigating (**m**) or even completely fending (**f**) the particular attacks. The intended countermeasure concepts are given as well. Note that the general countermeasure of performing access control is applicable to any of the attacks presented here, but it only mitigates the attack threat, and does not completely annihilate the possibility for an attack.

Amplification: This factor—as defined in [16]— is only applicable for flooding attacks and describes the relation of attack performance workload to attack impact workload. For example, in terms of message size related to memory usage, an amplification factor of 4 means that every byte of attack message data causes an allocation of 4 bytes of target system's memory.

6 Conclusion

Like every upcoming technology, Web Services are challenged by several security issues. The attacks presented in this article illustrate how easily an unsufficiently secured Web Service server can be affected with a single or few messages. While some of the vulnerabilities are caused by implementation weaknesses, most of them exploit fundamental protocol flaws, abusing the given flexibility within WS-related standards.

Thus, in order to cope with these threats, Web Service developers and adopters must be aware of the vulnerabilities and their potential impact. Further, researchers need to examine the existing Web Service standards for further vulnerabilities in order to develop more accurate countermeasures. Only improvement of attack mitigation techniques along with integration into every Web-Service-driven system will face up with these challenges and help to make Web Services as secure as possible.

References

1. Andrews T, Curbera F, Dholakia H, Goland Y, Klein J, Leymann F, Liu K, Roller D, Smith D, Thatte S, Tricko-

vic I, Weerawarana S (2003) Business Process Execution Language for Web Services Version 1.1. Oasis Standard
2. Bartel M, Boyer J, Fox B, LaMacchia B, Simon E (2002) XML-Signature Syntax and Processing. W3C Recommendation
3. Bhargavan K, Fournet C, Gordon AD, O'Shea G (2005) An advisor for Web Services security policies. In: SWS '05: Proceedings of the 2005 workshop on Secure web services, ACM Press, New York, NY, USA, pp 1–9
4. Fernando R (2006) Secure web services with apache rampart. Tech. rep., WSO2 Oxygen Tank
5. Gruschka N (2008) Schutz von Web Services durch erweiterte und effiziente Nachrichtenvalidierung. PhD thesis, Christian-Albrechts-University of Kiel, Germany
6. Gruschka N, Herkenhöner R (2006) WS-SecurityPolicy Decision and Enforcement for Web Service Firewalls. In: Proceedings of the IEEE/IST Workshop on Monitoring, Attack Detection and Mitigation
7. Gruschka N, Luttenberger N (2006) Protecting Web Services from DoS Attacks by SOAP Message Validation. In: Proceedings of the IFIP TC-11 21. International Information Security Conference (SEC 2006)
8. Gruschka N, Luttenberger N, Herkenhöner R (2006) Event-based SOAP message validation for WS-SecurityPolicy-Enriched web services. In: Proceedings of the 2006 International Conference on Semantic Web & Web Services
9. Gruschka N, Herkenhöner R, Luttenberger N (2007) Access Control Enforcement for Web Services by Event-Based Security Token Processing. In: Braun T, Carle G, Stiller B (eds) 15. ITG/Gi Fachtagung Kommunikation in Verteilten Systemen (KiVS 2007), pp 371–382
10. Gruschka N, Jensen M, Luttenberger N (2007) A Stateful Web Service Firewall for BPEL. Proceedings of the IEEE International Conference on Web Services (ICWS 2007)
11. Gudgin M, Hadley M, Rogers T (2006) Web Services Addressing 1.0 - SOAP Binding. W3C Recommendation
12. Hors AL, Hegaret PL, Wood L, Nicol G, Robie J, Champion M, Byrne S (2004) Document Object Model (DOM) Level 3 Core Specification. W3C Recommendation
13. Imamura T, Dillaway B, Simon E (2002) XML Encryption Syntax and Processing. W3C Recommendation
14. Jayasinghe D (2006) SOA development with Axis2: Understanding Axis2 basis. IBM developerWorks
15. Jensen M (2008) BPEL Firewall - Abwehr von Angriffen auf zustandsbehaftete Web Services (german). VDM Verlag Dr. Müller, ISBN 9783836485517
16. Jensen M, Gruschka N, Luttenberger N (2008) The Impact of Flooding Attacks on Network-based Services. In: Proceedings of the IEEE International Conference on Availability, Reliability and Security
17. Kaler C, Nadalin (editors) A (2005) Web Services Security Policy Language (WS-SecurityPolicy) 1.1
18. Leiwo J, Nikander P, Aura T (2000) Towards network denial of service resistant protocols. In: Proc. of the 15th International Information Security Conference (IFIP/SEC)
19. Lindstrom P (2004) Attacking and Defending Web Service. A Spire Research Report
20. McIntosh M, Austel P (2005) XML signature element wrapping attacks and countermeasures. In: SWS '05: Proceedings of the 2005 workshop on Secure web services, ACM Press, New York, NY, USA, pp 20–27
21. Nadalin A, Kaler C, Monzillo R, Hallam-Baker P (2006) Web Services Security: SOAP Message Security 1.1 (WS-Security 2004)
22. Needham RM (1994) Denial of service: an example. Commun ACM 37(11):42–46
23. Noga ML, Schott S, Löwe W (2002) Lazy XML processing. In: DocEng '02: Proceedings of the 2002 ACM symposium on document engineering, ACM Press, New York, NY, USA, pp 88–94
24. Schäfer G (2005) Sabotageangriffe auf Kommunikationsstrukturen: Angriffstechniken und Abwehrmanahmen. PIK 28 pp 130–139
25. Smith A (2007) Estonia: Under siege on the web. Time Magazine URL `http://www.time.com/time/magazine/article/0,9171,1626744,00.html`
26. The SAX Project (2002) Simple API for XML–SAX 2.0.1 URL `http://www.saxproject.org`
27. Weerawarana S, Curbera F, Leymann F, Storey T, Ferguson DF (2005) Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More. Prentice Hall PTR

Meiko Jensen studied computer science at the Christian-Albrechts-University of Kiel, Germany, and received his degree in computer science in 2007. Currently he is a Ph.D. student at the Horst Görtz Institute for IT-Security in Bochum, researching in the field of Web Service security, and is particularly interested in XML and service composition security, security modeling, and attacks on Web Services. Meiko is member of ACM and GI.

Nils Gruschka studied computer science at the Christian-Albrechts-University of Kiel, Germany, and received his Ph.D. in computer science in 2008. His thesis presents methods for Web Service protection using efficient Web Service message validation. Currently he works as a researcher scientist at NEC Laboratories Europe in the field of Web Service security. Nils is member of ACM and GI.

Ralph Herkenhöner studied computer science at the Christian-Albrechts-University of Kiel, Germany, and received his degree in computer science in 2006. Currently he is a Ph.D. student at the University of Passau, researching in the field of process and security modeling. In particular, he is interested in methodologies for proving privacy protection in business processes and IT environments. Ralph is member of ACM and GI.