

# Event-based SOAP Message Validation for WS-SecurityPolicy-Enriched Web Services

Nils Gruschka, Norbert Luttenberger, and Ralph Herkenhöner  
Communication Systems Research Group, Department for Computer Science  
Christian-Albrechts-University in Kiel, Germany  
{ngr|nl|rhk}@informatik.uni-kiel.de

**Abstract**—To enable checking of SOAP messages for compliance to a given security policy, extensions to the classical “Schema-only” validation of SOAP messages are required. These extensions check, if the WS-Security elements found in a SOAP message fulfill the Web Service security specification that is laid down in the WS-SecurityPolicy document. In this paper, we discuss to what extent the proposed extended validation of SOAP messages can be accomplished by an event-based validation system. We prefer this type of processing for use in network appliances like e.g. Web Service-level firewalls, because it is suited to resist DoS attacks that aim at memory exhaustion. We identify some of the constraints on the use of both WS-Security and WS-SecurityPolicy that must be introduced to allow for event-based parsing, and finally present an initial prototype for extended validation together with some performance figures.

**Keywords**—Web Services, WS-SecurityPolicy, SOAP message validation, event-based XML processing

## I. INTRODUCTION

In order to ensure the confidentiality, integrity, and authenticity of SOAP messages [1][2] exchanged between a Web Service client and a Web Service server, the Organization for the Advancement of Structured Information Standards (OASIS) has in 2004 proposed its WS-Security standard [3]. This by now widely accepted standard is built upon two underlying standards, namely XML-Encryption [4] and XML-Signature [5] that define data structures for encrypting and signing general XML documents. The WS-Security standard adopts these data formats and devises appropriate SOAP message format extensions, resulting in an extremely flexible security layout for SOAP messages.

In order to successfully apply any WS-Security primitives, Web Service clients and Web Service servers must obviously use pairwise matching policies on what elements of a SOAP message are to be encrypted and/or signed, and on what security tokens are to be supplied to each other. For this purpose, IBM, Microsoft and several others have in 2005 proposed a declarative XML-based language for security policies, called WS-SecurityPolicy [6]. Any Web Service server may thus lay down its security policy in WS-SecurityPolicy documents and provide them to clients. In a sense, a Web Service security policy thus can be considered as the non-functional companion to the functional Web Service description as defined in the WSDL [7] document that comes with every Web Service.

Evidently, the WS-Security primitives discussed so far are of no help to improve the *availability* of Web Services.

On the contrary: it must be admitted that complex security primitives downright provoke new kinds of Denial-of-Service (DoS) attacks that are suited to engage a server in computations that consume large amounts of the available processor and memory capacity. This is due to the fact that WS-Security combines intense XML parsing and validation with cryptographic computations. Especially, the latter offer new possibilities for attacks [8][9], mainly caused by the high computation cost for cryptographic functions. A simple kind of attack might e.g. involve a server in lengthy computations on malformed messages with invalid security tokens—resulting in “only” a message refusal. Consequently, WS-Security needs an elaborate distribution of security and server functions enabling a sound balance between all mentioned security goals.

A suitable network configuration—as e.g. brought forward by DataPower under the term “XML-Aware Network” (XAN) infrastructure—offloads message validation to network appliances located “in front of” the Web Service server. In a similar effort [10], we showed how SOAP message schemas can be derived from WSDL-based Web Service descriptions and how from these schemas, XML-aware firewall systems can be constructed that validate incoming and outgoing SOAP messages. Validation in that approach is restricted to schema validation. In this paper, we aim at extending validation to handle WS-Security tokens as defined by a service’s WS-SecurityPolicy specifications. We call this an *extended validation*. Extended validation thus has to check any SOAP message against both functional and non-functional specifications.

Performance is an important imperative for firewalls: a firewall should neither noticeably increase a server’s response time, nor must a firewall contribute to an attacker’s success by lack of processing power. This argument advocates event-based XML processing following the SAX paradigm [11]. SAX-type parsers do not construct a complete programming language-based internal XML document representation in main memory, but instead process an XML document step-by-step. In the ideal case, i.e. when no elements from the parsed XML document have to be stored for later processing, event-based parsing thus does not consume any memory. Additionally—what is very important for the type of application we have in focus here—a validating event-based parser aborts processing when it encounters an invalid XML element. In this paper, we discuss the problem to what degree an extended validation in the above defined sense can be

performed by event-based XML processing.

This article is organized as follows: We start with related work on XML firewalls and Web Service policies. In section III we present some threats for Web Services. After that we give a short introduction on the WS-Security and WS-SecurityPolicy. The following section shows our concept for event-based extended validation. Section VI presents and evaluates our prototype implementation. The article closes with a summary and an outlook.

## II. RELATED WORK

Firewall systems working below the “XML layer”—i.e. packet filters and application level gateways—have been extensively discussed for years. For Web Services and *XML-Aware Networks*, firewalls operating on the content of SOAP messages are of essential importance. In [10] we have presented a Web Service firewall that validates incoming and outgoing SOAP messages against an XML Schema, derived from the Web Service description. Similar approaches are made by commercial Web Service firewalls like Forum Systems XWall. However, these firewalls are unable to check the correctness of WS-Security tokens, i.e. verify signature values, process encrypted document parts, etc. In no case are they able to check whether the security tokens indeed fulfill the requirements specified by the appropriate WS-SecurityPolicy document.

Research on WS-SecurityPolicy is mostly limited to modelling and verifying security policies. In [12] an example shows, how policies for Web Services can be modelled from a business-oriented view using well experienced patterns for security requirements. In [13] and [14] cryptographic protocols defined by a security policy are analyzed for vulnerabilities to *XML rewriting attacks*, i.e. if the allowed SOAP message can be maliciously intercepted or manipulated despite of security tokens.

To our knowledge, this is the first approach to WS-SecurityPolicy and WS-Security processing using an event-based processing approach.

## III. THREATS FOR WEB SERVICES

The purpose of a Web Service firewall is to protect the Web Service server from threats coming from any kind of “malicious” SOAP messages<sup>1</sup>. These might be messages not conforming to the Web Services specification or even messages, that are part of a DoS attack. The following list gives an overview of threats for a Web Service server.

**Non-Web-Service-description conforming message:** A Web Service defines appropriate SOAP message formats in a Web Service description using the WSDL (see above). Any SOAP message not satisfying the devised message grammar must be rejected by a Web Service server. In order to protect a server from wasting valuable processing time on such a message, it has to be rejected by the Web Service firewall.

<sup>1</sup>Protection from attacks using protocols below the XML layer is performed by (additional) packet filters or HTTP gateways and are beyond the scope of this article.

Thus a fundamental function for a Web Service firewall is checking a SOAP message’s conformance to the Web Service description.

**Oversize Payload:** An *Oversize Payload* [15] attack is a kind of *Resource Exhaustion* [8] attack using 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. When using DOM (which is the most common XML processing model) the parser reads the complete SOAP message and builds an in-memory representation (called DOM tree) that is much larger than the message itself. Checking if a message fulfills the service’s interface can not start until the document is completely read. Thus possibilities for DoS attacks using extremely large arbitrary SOAP messages are opened. Further on, even with other processing models than DOM, possibilities for the above mentioned attacks remain, e.g. by using weaknesses in the Web Service description’s types section (see [10]).

**WSDL Scanning:** Even if only some of a service’s operations should be accessible from the outside network (and only those that are enumerated in the published Web Service description), an attacker can try downright to guess some of the other operations’ names and to call them. Packet filters and HTTP application level gateways are in most cases unable to differentiate operations belonging to the same service, due to the fact that they all have the same service endpoint (IP address, TCP port and HTTP URL) and therefore do not reject these messages.

When adding security tokens and security policies to a Web Service communication, additional threats arise.

**Non-policy conforming message:** As presented above, the Web Service policy defines the required security elements for the SOAP messages, i.e. which document parts have to be signed, which have to be encrypted, which tokens have to be used, etc. All messages not complying to this policy are invalid and must be rejected.

**Modified message:** XML-Signature is used to ensure the integrity of specific parts of the SOAP message. To detect modification of the SOAP message, the signatures demanded by the security policy must be verified. Again: an attacker could engage a server into costly signature verifications on invalid signatures and thereby reduce the Web Service server’s capacity for useful work.

**Encrypted large blocks:** As stated above, one possibility for detecting an *Oversize Payload* attack (inside the functional message part) is by validating the SOAP message to a modified Schema. However, if this message part is encrypted, validating can not be performed until it is decrypted. If the encrypted part then is completely loaded into memory before processing the encryption (which is required for DOM processing), the *Oversize Payload* attack would be successful, regardless of whether the validating component is able to fend it or not.

These threats led us to a extended definition of “validation” and to the conclusion that the DOM processing model is inappropriate for network appliances serving security purposes. Before discussing our event-based validation solution,

we give a short introduction on the WS-Security and WS-SecurityPolicy specifications, which are the foundation of our validation model.

#### IV. WS-SECURITY AND WS-SECURITYPOLICY

Figure 1 shows the relationship of the standards involved in secure Web Services.

	<i>functional</i>	<i>non-functional</i>
<i>meta specification</i>	WSDL	WS-SecurityPolicy
<i>specification</i>	SOAP	WS-Security

Fig. 1. Web Service specifications

WS-Security [3] extends the SOAP specifications by defining security tokens for security features, mainly integrity, confidentiality and authentication for Web Service messages. WS-Security reuses existing XML security standards like XML-Signature [5] and XML-Encryption [4] to reach these security goals.

The sample SOAP message in figure 6 contains: a `<BinarySecurityToken>`, which holds an X.509 certificate for cryptographic operations; an `<EncryptedKey>`, which holds the symmetric key—encrypted with the receivers public key—used for the encrypted body content; a `<Signature>`, signing the body and the timestamp; a timestamp, used to avoid replay attacks; and finally an `<EncryptedData>` block, holding the SOAP body (see 7) in encrypted form.

WS-SecurityPolicy [6] provides an XML syntax for declarative security policies. Like a Web Service description defining the allowed SOAP messages, a WS-SecurityPolicy policy defines the required WS-Security elements.

A security policy is composed from *Assertions* combined by operators `<All>` (which may also be expressed by `<Policy>`) and `<ExactlyOne>`. WS-SecurityPolicy defines a number of assertions, the most important ones are *Protection Assertions*, *Token Assertions* and *AlgorithmSuite Assertions*. In figure 9 two Protection Assertions, namely `<SignedElements>` and `<EncryptedParts>`, are shown. In figure 8 a Token Assertion and an Algorithm Assertion are given.

A Web Service policy may consist of multiple WS-SecurityPolicy documents for different subjects of the Web Service. Figure 8 shows a sample endpoint policy, figure 9 a sample message policy.

In detail, the message policy requires that the SOAP body is signed and the operation `<payOrder>` (addressed by the XPath expression `/Envelope/Body/payOrder`) is encrypted. The endpoint policy demands an X.509 token used for protection (i.e. encryption and signature operations), a timestamp and the use of specific algorithms for the cryptographic functions. The sample SOAP message in figure 6 complies to the accumulated security policy.

##### A. XML-Encryption

XML-Encryption is a W3C standard for the encrypting arbitrary digital content. Following this standard, encrypted content is wrapped by an `<xenc:EncryptedData>` XML

```
<xenc:EncryptedData Id=...>
  <xenc:EncryptionMethod Algorithm=.../>
  <ds:KeyInfo>
    ...
  </ds:KeyInfo?>
  <xenc:CipherData>
    <xenc:CipherValue?>
    <xenc:CipherReference URI??>
  </xenc:CipherData>
</xenc:EncryptedData>
```

Fig. 2. Simplified XML-Encryption Grammar (excerpt)

```
<ds:Signature>
  <ds:SignedInfo>
    <ds:CanonicalizationMethod Algorithm=.../>
    <ds:SignatureMethod Algorithm=.../>
    <ds:Reference URI=...>
      <ds:Transforms>...</Transforms?>
      <ds:DigestMethod Algorithm=.../>
      <ds:DigestValue>...</DigestValue>
    </ds:Reference> +
  </ds:SignedInfo>
  <ds:SignatureValue>...</SignatureValue>
  <ds:KeyInfo>...</KeyInfo?>
</ds:Signature>
```

Fig. 3. Simplified XML-Signature Grammar (excerpt)

element (see figure 2). An `<xenc:EncryptedData>` element specifies the encryption algorithm, allows the receiver to determine the decryption key, and encloses the encrypted data (“enveloping”) or a reference to these (“referencing”).

WS-Security defines that each part of a SOAP message, which is appointed for encryption, is replaced by the sender by an `<xenc:EncryptedData>` element. Inside SOAP messages, only the enveloping mode must be considered. WS-Security additionally allows SOAP messages to enclose one or more `<xenc:EncryptedKey>` elements as defined in XML-Encryption, that have a structure similar to the structure of `<xenc:EncryptedData>` elements. For decryption of encrypted message parts a stream-based procedure as proposed in [16] can be used. To re-transform an `<xenc:EncryptedData>` element into its original form, the following steps have to be performed:

- 1) Determine the decryption key (This possibly implies to resolve an `<xenc:EncryptedKey>` element.).
- 2) Process any information from the `<xenc:EncryptedData>` element that is related to the applied encryption method
- 3) Decrypt the enveloped encrypted data
- 4) Replace the `<xenc:EncryptedData>` element by the decrypted data

##### B. XML-Signature

XML-Signature [5] is a W3C recommendation for digital signatures using an XML format. It is applied by WS-Security for message authentication and ensuring message integrity.

Figure 3 shows a simplified schema for an XML-Signature. The `<ds:Signature>` element contains the following three elements: a `<ds:SignedInfo>` element holding—besides

some technical declarations—references to the objects—in this context: message elements—covered by the signature, a `<ds:SignatureValue>` element holding the digital signature over the `<ds:SignedInfo>` element and a `<ds:KeyInfo>` element declaring the signing key. Each `<ds:Reference>` contains a reference to the signed object using either Shorthand XPath References [17] or an XPath Filter 2.0 Transform. Furthermore, it contains a digest of the respective message fragment. To verify an XML-Signature the referenced element must be detected, canonicalized, digested and compared to the given digest value. Finally the `<ds:SignedInfo>` signature has to be verified.

The following section discusses how to include WS-Security and WS-SecurityPolicy into an “extended” validation and to what degree this validation can be based on an event-based processing model.

## V. EVENT-BASED EXTENDED VALIDATION

### A. Concept

Following the classical definition, XML document validation encompasses a well-formedness check and a check against a document grammar, the latter nowadays usually defined in an XML Schema document [18]. For WS-Security-enriched SOAP messages this kind of validation is unfortunately no longer sufficient: Additionally it must be checked if a SOAP message fulfills the Web Service server’s security policy (laid down in a WS-SecurityPolicy document). We denote this kind of validation “extended validation”. As stated earlier, our goal is to find out, what restrictions must be applied to WS-Security and WS-SecurityPolicy in order to perform an extended validation in an event-based parsing and validation approach.

Extended validation can be decomposed into the following building blocks:

- examination of the security tokens sent within a SOAP message (presence and type): required by the Web Service server’s security policy
- examination of encrypted and signed message parts (presence): required by the Web Service server’s security policy
- verification of digital signatures: required for establishing SOAP message integrity and authenticity
- decryption of encrypted message parts: required for classical validation
- classical validation of the SOAP message.

At first glance, this list seems to propose a two-pass processing of SOAP messages: Pass #1 would handle all security aspects and on a positive outcome would remove all related message parts, pass #2 would in classical manner validate the “purified” SOAP message against the message schema as contained in the Web Service description. However, it is obvious, that a two-pass processing strategy violates the event-based “on-the-fly” parsing and validation approach: The complete message had to be kept in memory after pass #1 to forward it to pass #2.

The question remains, what is the correct order to chain up the given building blocks. In order to avoid costly cryptographic computations, one could propose to postpone signature verification and message decryption until immediately before classical validation. This seems to be a reasonable approach: A message that does not fulfill the given security requirements could be dropped without wasting valuable processor capacity for cryptographic operations. Assume a Web Service operation containing some sort of critical input like a credit card number (see figure 7). The Web Service operator would be right to require encryption for this input and to drop any message not obeying this policy, if even other parts of those messages were correctly encrypted and/or signed.

Unfortunately, this approach does not work. The reason is simple: Checking if the sender encrypted those parts of a message that were actually determined to be encrypted by the security policy requires to decrypt the encrypted message parts. I.e. for checking a message against a security policy, decryption is required and cannot be postponed. In the sample above the policy requires the element `<payOrder>` to be encrypted (see figure 9), which can only be tested if the element `<xenc:EncryptedData>` (in figure 6) is resolved and its content decrypted.

The resulting chain of validation steps is shown in figure 4. Clearly, cryptographic operations are carried out before any validations against grammar or policy are performed.

This approach also raises a number of issues for processing of XML-Encryption, XML-Signature and WS-SecurityPolicy, discussed in the following paragraphs.

### B. XML-Encryption Processing

A pure event-based decryption/validation procedure would run into a problem, if the decryption key were specified by a forward reference to an `<xenc:EncryptedKey>` element further in the SOAP message, which in principle is allowed by XML-Encryption. This would force the validator to cache a complete block of encrypted data with no meaningful size limit, until the validator meets the required `<xenc:EncryptedKey>` element. This would invite attacks aiming at memory exhaustion. Therefore, we exclude this case: “key first” is no functional constraint, and is prescribed in the interoperability requirement R3208 of [19], too.

The content of the `<xenc:CipherValue>` element can now be decrypted “on the fly”, because all permitted encryption algorithms [4] are block ciphers. The encrypted data stream can be forwarded to a SAX parser, and the resulting events can be validated e.g. against the SOAP message schema as specified by the Web Service description.

### C. XML-Signature Processing

Event-based signature processing works roughly speaking in this way: when a `<ds:Signature>` element event appears, the signature fragment is completely read and the signature can be verified. It remains the task of calculating the digest values for the references. If the reference refers “forward” the digest

value and the reference are stored until the referred element is read and the digest is calculated.

But the reference can as a matter of fact be a backwards reference, like in the sample message in figure 6: the second reference inside the signature points to the timestamp element (using the XPointer `#timestamp`) at the beginning of the SOAP message. With event-based processing there is no possibility to “go back” inside the document. Hence it is of necessity to digest the signed element before appearance of the `<ds:Signature>` element.

A possible solution is calculating a digest for all nodes inside a document. In [20] an implementation using this procedure is presented. One can prove that such a solution has a memory consumption of  $O(n^2)$  with  $n$  being the number of XML nodes inside the message. Since high memory consumption increases the vulnerability for DoS attacks, this is not an acceptable approach.

For a viable solution, we propose to restrict the possibilities for referencing a signed element: A signed element should be referenced by XPointer only. This is the recommended method for referring to a signed element in R3001 of [19]. There is only one problem with this solution, namely the necessity of an identifying ID attribute in the referenced element. However, most XML documents allow additional attributes from different namespaces, thus in most cases it is possible to add a `wsu:Id="identifier"` attribute for referencing the element. In figure 6 this was done for the signature reference to the SOAP body: the body element contains an attribute `wsu:Id="body"` and the signature refers to the body using the Shorthand XPointer `URI="#body"`.

Using this restriction, only those elements containing an ID attribute have to be digested. The actual canonicalisation and digesting operation can be executed event-based trouble-free—besides some smaller problems explained in [21].

Thus, event-based signature verification is possible if only Shorthand XPointer References are used.

#### D. WS-SecurityPolicy Processing

Checking compliance of a SOAP message to a security policy is very complex; presenting all details is beyond the scope of this article. We just present three of the problems that emerge in this process.

- 1) Extracting security relevant parsing events from security policies requires scanning of all security assertions. These are spread over a number of policy documents dealing with different security subjects. In the sample policy of figures 7 and 8 the assertions `<SignedParts>` `<Body>`, `<ProtectionToken>` `<WssX509V3Token10>` and `<AlgorithmSuite>` `<TripleDesRsa15>` collapse into the requirement that the SOAP message body has to be signed with SHA-1 and RSA using an X.509 token.
- 2) Security assertions can be combined by “and” and “or” inside a policy. Checking if all requirements are fulfilled is equivalent to the boolean satisfiability problem (SAT), which is known to be exponential in runtime. To reduce

complexity, the policy can however be converted into a normalized form (for details see [21]) and represented as a set of all possible requirements. This increases memory consumption for complex policies, but reduces processing time for practical applications considerably.

- 3) A further problem for checking SOAP message compliance arises from XPath expressions in policy assertions. If two assertions contain different XPath expressions sharing a non-empty intersection, it cannot be decided to which of the assertions a node from inside the intersection has to be mapped. A similar problem arises for mapping security tokens from SOAP messages to policy requirements. This is caused by the imprecise token specification in WS-SecurityPolicy. For both mapping problems, “best fit” is a pragmatic solution. Criteria for “best fit” are token type, usage algorithm and reference of the applied element.

For an extended validator, the earliest opportunity for checking policy compliance arrives when the parser encounters the end of a security header (`</wsse:Security>`). Thus, message processing can be stopped before the end of a SOAP message is reached. At other points of the SOAP message, the compliance to the policy is usually undecidable.

## VI. PROTOTYPE IMPLEMENTATION

In [10] we presented *CheckWay*, a firewall system for non-WS-Security-enriched Web Services. *CheckWay* validates all incoming SOAP messages against an XML Schema [22], derived from the Web Service description. All non-valid messages are dropped by the firewall, only valid message are forwarded to the Web Service server. We have shown how message validation plus “hardening” the Web Service Schema increase the robustness against *Denial of Service* (DoS) attacks.

This system was extended to *CheckWaySec* implementing a system for extended event-based validation. SOAP messages are checked additionally on complying to a Web Service’s security policy. If the message is invalid—in terms of the definition of *extended validation* given before—it is dropped by *CheckWaySec*. This system can either be used as pre-processor for legacy Web Service systems, not capable of WS-Security/WS-SecurityPolicy processing or stand-alone in SOAP intermediaries or XML firewalls.

Figure 4 shows the overall architecture for *CheckWaySec*. The system works roughly speaking the following way:

- The incoming SOAP messages are parsed by the SAX Parser into SAX events.
- The Signature Handler analyzes the `<Signature>` fragments, computes digest values for (potentially) signed parts and verifies the signature.
- The Encryption Handler analyzes the `<EncryptedKey>` fragments and extracts the keys needed for decryption. Further on it decrypts all encrypted parts. The decrypted document part is again forwarded to the SAX parser (not shown in the figure).

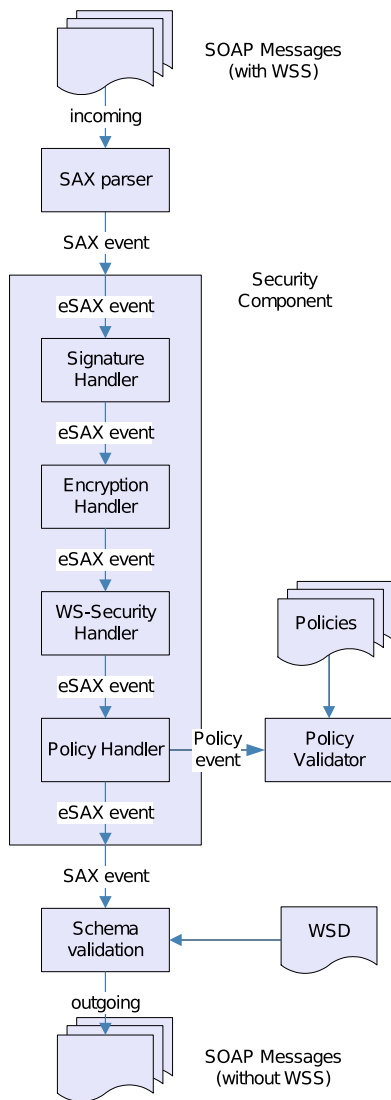


Fig. 4. Architecture for Extended Validation

- The WS-Security Handler processes the remaining security tokens, e.g. timestamps, binary security tokens, etc.
- The Policy Handler collects all WS-Security relevant events and generates policy events for the Policy Validator.
- The Policy Validator validates the policy events to the security policy. It uses a special internal presentation for the security policy generated from policies written in WS-SecurityPolicy.
- The Schema Validator finally validates the whole SOAP message (without the WS-Security tokens consumed by the other handlers) against an XML Schema derived from the Web Service description (WSD) (see [10]).

This leads to a continuous event-based processing. If any component from this handler chain detects a violation inside a SOAP message event, an exception is thrown back and processing of this message is stopped. The SOAP message will either just be dropped or a SOAP fault message is sent to the message's sender.

The event-based architecture leads to higher performance especially for invalid message, because a message is only read and processed until the element causing the violation, and low memory consumption, because it is not necessary to store the whole document. Only values needed ahead in the document are stored (in brackets the counterpart in the sample message in figure 6 is given):

- tokens for usage with cryptographic functions (C)
- decrypted keys for encryption of <EncryptedData> (D and G)
- digests of (potentially) signed blocks for comparing with digest value inside a <Signature> (*backward reference*, B and E)
- digest values from <Signature> elements for comparing with digest of signed blocks (*forward reference*, B and F)
- abstract information about security tokens for policy checks (see below)

Inside the security component the event communication is performed via an *extended SAX* interface (eSAX). It adds events for transport of WS-Security and WS-SecurityPolicy relevant information to the standard SAX interface. As an example the Encryption Handler consumes the standard SAX events

```

startElement("EncryptedData")
...
endElement("EncryptedData")
and transmits the extended SAX events
consumedStartElement("EncryptedData")
...
consumedStartElement("CipherValue")
startElement("payOrder")
...
endElement("payOrder")
consumedEndElement("CipherValue")
...
consumedEndElement("EncryptedData")

```

As mentioned before, the Policy Validator generates policy events from these eSAX events. The complete sequence of policy events for the example in figure 6 is:

```

startMessage()
startSecurityHeader()
timestamp("2005-08-06T21:20:00Z",
"2006-08-06T21:20:00Z")
signedReference("X509Token", "...#rsa-sha1",
"...c14n#", "/Envelope/Header/Security/TimeStamp")
securityToken("X509Token", "...#X509v3", "instance")
keyWrapping("X509Token", "...#rsa-1_5")
securityToken("X509Token", "...#X509v3", "reference")
endSecurityHeader("strict")
encryptedReference("X509Token", "...#tripledes-cbc",
"/Envelope/Body/payOrder")
signedReference("X509Token", "...#rsa-sha1",
"/Envelope/Body")
endMessage()

```

These events are used by the Policy Validator to check if the security policy is fulfilled. As an example the policy event `signedReference("X509Token", "...#rsa-sha1", "/Envelope/Body/payOrder")` is mapped to the

requirement, that the SOAP body has to be signed by an X.509 token using RSA and SHA-1.

For evaluation purposes we performed a runtime and memory usage experiment using a policy similar to the sample included in this paper (i.e. signed and encrypted SOAP body). We created SOAP messages of increasing length (up to 500,000 XML elements) by varying the number of elements inside the encrypted part. We conducted two test series: one series with messages complying to both policy and message grammar, and another with messages violating the message grammar. The grammar violation occurred in the beginning of the encrypted message part.

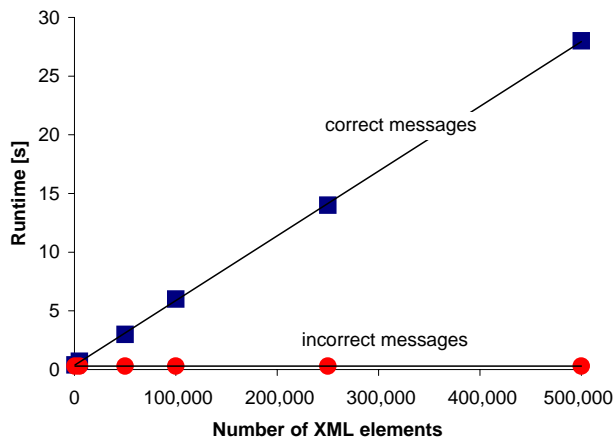


Fig. 5. Validation runtime

Figure 5 shows linear increasing runtime for correct messages and constant runtime for incorrect messages. This behaviour shows the advantages of event-based extended validation: message processing for incorrect messages is always terminated as early as possible while maintaining linear runtime for correct messages.

The memory consumption was observed to be independent of message correctness and message sizes (approx. 9 MByte).

When using a DOM-based decryption engine (Apache XML Security, Java) we experienced a memory consumption of 64 MByte for a message of 80,000 elements and out-of-memory exceptions for messages larger than 85,000 elements.

## VII. SUMMARY AND OUTLOOK

In this article we have discussed the potential possibilities and the limitations of processing SOAP messages containing WS-Security tokens, defined by a WS-SecurityPolicy policy using an event-based system. Proof has been given that such an approach only requires some non-critical restrictions. On the other hand, event-based processing results in more efficient SOAP processing in regard to time and memory consumption. This efficient processing combined with validating against Web Service description and security policies leads to a system for protecting Web Service servers from invalid messages and therewith a variety of denial of service attacks.

Protection of Web Services from invalid messages is limited by too lax expressed specifications. For example, WS-SecurityPolicy contains only little “maximum restrictions”. A message containing additional unused tokens is still valid, but engages the server in needless computation. Further on, XML Schemas for security tokens contain a lot of extension points, allowing too many possible messages.

Nevertheless, event-based processing for WS-Security and WS-SecurityPolicy increases performance for security-enabled SOAP message processing and this way the availability of Web Service servers.

## REFERENCES

- [1] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, and H. F. Nielsen, “SOAP Version 1.2 Part 1: Messaging Framework,” *W3C Recommendation*, 2003.
- [2] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard, “Web Services Architecture,” *W3C Recommendation*, 2004.
- [3] B. A. et al., “Web Services Security (WS-Security),” 2002.
- [4] T. Imamura, B. Dillaway, and E. Simon, “XML Encryption Syntax and Processing,” *W3C Recommendation*, 2002.
- [5] M. Bartel, J. Boyer, B. Fox, B. LaMacchia, and E. Simon, “XML-Signature Syntax and Processing,” *W3C Recommendation*, 2002.
- [6] G. D.-L. et al., “Web Services Security Policy Language (WS-SecurityPolicy),” 2005.
- [7] E. C. et al., “Web Services Description Language (WSDL),” *W3C Note*, 2001.
- [8] G. Schäfer, “Sabotageangriffe auf Kommunikationsstrukturen: Angriffstechniken und Abwehrmaßnahmen,” *PIK 28*, pp. 130–139, 2005.
- [9] C. Meadows, “A Formal Framework and Evaluation Method for Network Denial of Service,” in *PCSFW: Proceedings of the 12th Computer Security Foundation Workshop*, 1999.
- [10] N. Gruschka and N. Luttenberger, “Protecting Web Services from DoS Attacks by SOAP Message Validation,” 2006.
- [11] The SAX Project, “Simple API for XML – SAX 2.0.1,” 2002.
- [12] M. Tatsubori, T. Imamura, and Y. Nakamura, “Best-practice patterns and tool support for configuring secure web services messaging,” in *ICWS '04: Proceedings of the IEEE International Conference on Web Services (ICWS'04)*. Washington, DC, USA: IEEE Computer Society, 2004, p. 244.
- [13] K. Bhargavan, C. Fournet, and A. D. Gordon, “Verifying policy-based security for Web Services,” in *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*. New York, NY, USA: ACM Press, 2004, pp. 268–277.
- [14] K. Bhargavan, C. Fournet, A. D. Gordon, and G. O’Shea, “An advisor for web services security policies,” in *SWS '05: Proceedings of the 2005 workshop on Secure web services*. New York, NY, USA: ACM Press, 2005, pp. 1–9.
- [15] P. Lindstrom, “Attacking and Defending Web Service,” *A Spire Research Report*, 2004.
- [16] T. Imamura, A. Clark, and H. Maruyama, “A Stream-based Implementation of XML Encryption,” 2002.
- [17] S. DeRose, E. Maler, and R. Daniel Jr., “XML Pointer Language (XPointer) Version 1.0,” *W3C Last Call Working Draft*, 2001.
- [18] E. van der Vlist, *XML Schema*. O’Reilly, 2002.
- [19] A. B. et al., “Basic Security Profile Version 1.0,” *WS-I Organisation*.
- [20] W. Lu, K. Chiu, A. Slominski, and D. Gannon, “A Streaming Validation Model for SOAP Digital Signature,” 2004.
- [21] R. Herkenhöner, “Eventbased and Policy-driven Web Service Firewall,” *Diploma thesis*, 2005.
- [22] H. T. et al., “XML Schema Part 1: Structures Second Edition,” *W3C Recommendation*, 2004.

<env:Envelope> <env:Header>	A
<wsse:Security> <wsu:Timestamp Id="timestamp"> <wsu:Created>...</wsu:Created> <wsu:Expires>...</wsu:Expires> </wsu:Timestamp>	B
<wsse:BinarySecurityToken Value="wsse:X509v3" Id="X509Token"> MIEZzCCA9CgAwIBAgIQEmtJZc0rqrKh5i... </wsse:BinarySecurityToken>	C
<xenc:EncryptedKey> ... <ds:KeyInfo> <wsse:SecurityTokenReference> <wsse:Reference URI="#X509Token" /> </wsse:SecurityTokenReference> </ds:KeyInfo> <xenc:CipherData> ... </xenc:CipherData> <xenc:ReferenceList> <xenc:DataReference URI="#enc"/> </xenc:ReferenceList> </xenc:EncryptedKey>	D
<ds:Signature> <ds:SignedInfo> ... <ds:SignatureMethod Algorithm="...#rsa-shal"/> <ds:Reference URI="#body"> ... <ds:DigestValue> XJb0lm4bHp1ZwIB... </ds:DigestValue> </ds:Reference> <ds:Reference URI="#timestamp"> ... <ds:DigestValue> LyLsF094hPi4wPU... </ds:DigestValue> </ds:Reference> </ds:SignedInfo> <ds:SignatureValue> Hp1ZkmFZ/2kQLXDJbchm5gK... </ds:SignatureValue> <ds:KeyInfo> <wsse:SecurityTokenReference> <wsse:Reference URI="#X509Token"/> </wsse:SecurityTokenReference> </ds:KeyInfo> </ds:Signature>	E
</wsse:Security> </env:Header> <env:Body wsu:Id="body">	F
<xenc:EncryptedData Id="enc"> <xenc:EncryptionMethod Algorithm="...#tripledes-cbc"/> <xenc:CipherData> <xenc:CipherValue> d2FpbmdvbGRfE0lm4byV0... </xenc:CipherValue> </xenc:CipherData> </xenc:EncryptedData>	G
</env:Body> </env:Envelope>	H

Fig. 6. Sample SOAP message with WS-Security tokens

```

...
<env:Body Id="body">
  <ns:payOrder>
    <orderID>
      XYZ-12345
    </orderID>
    <creditCardNumber>
      1234 5678 9012 3456
    </creditCardNumber>
  </ns:payOrder>
</env:Body>
...

```

Fig. 7. (Unencrypted) SOAP body inside the sample message

```

<wsp:Policy>
  <sp:SymmetricBinding>
    <wsp:Policy>
      <sp:ProtectionToken>
        <wsp:Policy>
          <sp:X509Token>
            <wsp:Policy>
              <sp:WssX509V3Token10/>
            </wsp:Policy>
          </sp:X509Token>
        </wsp:Policy>
      </sp:ProtectionToken>
      <sp:AlgorithmSuite>
        <wsp:Policy>
          <sp:TripleDesRsa15/>
        </wsp:Policy>
      </sp:AlgorithmSuite>
      <sp:Layout>
        <wsp:Policy>
          <sp:Strict/>
        </wsp:Policy>
      </sp:Layout>
      <sp:EncryptBeforeSigning/>
      <sp:IncludeTimestamp/>
    </wsp:Policy>
  </sp:SymmetricBinding>
</wsp:Policy>

```

Fig. 8. Sample endpoint policy

```

<wsp:Policy>
  <sp:EncryptedElements>
    <sp:XPath>
      /Envelope/Body/payOrder
    </sp:XPath>
  </sp:EncryptedElements>
  <sp:SignedParts>
    <sp:Body/>
  </sp:SignedParts>
</wsp:Policy>

```

Fig. 9. Sample message policy