Web Services Security

Renaud Bidou

DENYALL

rbidou@denyall.com

Abstract

Web Services are getting more and more widely deployed as the adoption of XML, SOAP and other related standards provides a generic and flexible framework for the implementation of Service Oriented Architectures (SOA).

These architectures are designed to provide automated interactions between data and processes, speed up business collaboration and ease the interconnection of formerly heterogeneous applications. Getting rid of the human operations, which could either be considered as a bottleneck, a single point of failure or an additional source of errors, is one of the main advantages of SOA.

However the capability to design open, distributed and automated service chains raises several security challenges which can be split in two categories.

The first one is the digital translation of security issues already present in the previous, human-based, organizations. Such issues are mostly related to information leak, leading to the identification of unknown contact points, weaknesses in control processes or disclosure of confidential information.

The second challenge to face is the introduction of new technologies, and consequently of new vulnerabilities and attack vectors. While flue and holidays have been removed from the chain, new business stoppers have appeared.

Moreover new technologies bring their load of confusion, usually leveraged by the vendors' will to push their own solutions and save their R&D investment. This situation reaches a peak with web services as they intend to replace most of the common components involved in the communication chain between applications.

The purpose of this document is first to provide a clear and objective overview of the security challenges faced by web services.

We will first identify the main components of a web service infrastructure and clarify their roles and purpose. In the second part we shed the lights on existing security standard which deserve specific attention. Then we will focus on the threats such an infrastructure is exposed to, by detailing attack techniques and their impact on the security of the targeted services.

Web Services Infrastructure

Protocols and languages

In order to provide openness and interoperability, web services need to rely on a set of protocols and languages likely to be implemented on all the systems which would be involved in the application chain.

Therefore standards have been adopted and officially recommended for implementation in the web service infrastructure. Some other have been created in order to match some functional needs which were not covered yet by existing standards.

And this is XML which has been accepted as the unique standard foundation for all languages and communication protocols in use in the web services infrastructure.

On the communication side SOAP, an XML based communication protocol, has need defined as the standard protocol for any kind of communication between the components of the web service infrastructure, would they be involved in data processing, security, control or any other function.

XML et Al.

Definition of XML

XML is a W3C[1] recommendation. Current version is 1.1 [XML11] which 2nd edition has been published on August 16th, 2006.

XML (eXtensible Markup Language) has been originally designed to store and transmit structured text data. Its main goal is to ease communication and interoperability between heterogeneous systems. As such it can be considered as a generic framework providing a unified syntax and a common structure for any kind of data.

An XML document can be represented as a tree with one unique root and any number of branches and leaves. These components are named nodes and can be of four types:

- root: a unique node represented with the "/". All nodes are descendants of the root;
- elements: named components which can support any kind of node: attributes, text or elements;
- attributes: are (name,value) couples, unique per element. They are used to qualify an element;
- text: these are nodes which don't have children and are always contained into an XML element. They can support many kind of encoding and therefore provide the necessary flexibility for data representation.

The openness of the XML structure has led to its wide adoption, either by commercial organisation and the open source community. Its syntax is used by several languages and protocols such as XHTML, XSLT and SOAP.

xPath and xQuery

xPath

xPath is a W3C recommendation. Current version is 2.0 [XPATH20] which has been published on January 23rd, 2007.

xPath is a language which makes it possible to search pieces of information into an XML document. It is mainly used to browse into the structure of such documents and is the base components of other languages such as xQuery and xPointer. As a consequence it is to be considered as one of the main block of advanced XML exploitation mechanism.

[1] Wolrd Wide Web Consortium (W3C) : http://www.w3.org

Elements or group of elements are selected, manipulated and compared through the use of "path expressions". xPath defines different types of relationships between elements.

- Parents: every node but the root has one parent;
- Children: node can have no, one or more children;
- Siblings: nodes which have the same parent are siblings;
- Ancestors: all nodes hierarchically higher than the node;
- Descendants: all nodes hierarchically lower than the node

Main components of path expressions are provided in the table below.

Node name	Selects all the children of the node.
/	Selects all elements from the root
//	Selects all elements from the named node whatever their position is
.	Selects the current node
..	Select the parent of the current node
@	Selects attributes

Table 1: xPath expressions components

In order to filter-out elements retrieved through a path expression, xPath makes use of predicates which are provided between square brackets.

xPath also provides the ability to select multiple paths thanks to the |operator which performs a logical AND operation between expressions.

Examples

home: selects all the children of the element <home>

/home: selects the element <home>

/home/room: selects all the elements <room> which are the children of the element <home>

//room: selects all the elements <room> wherever they are in the XML documents

//room@size: selects all the attribute <size> of all the elements <room> into the diocument

/home/room[1]: selects the first <room> element which is a child of the <home> element

/home/room[@size>15]: selects all the <room> elements whose attribute <size> is greater than 15

/home/room[@size>15]/name: selects the all the elements <name> which are the children of <room> elements selected as specified above

/home/room[@size>15] | /home/garden: selects all the <room> elements whose attribute <size> is greater than 15 AND the elements <garden> which are the children of the element <home>

xQuery

xQuery is a W3C recommendation. Current version is 1.0 [XQUERY10] which has been published on January 23rd, 2007.

xQuery could be considered as the SQL equivalent for XML. It shares the same data model, functions and operators with xPath. Howhever it noticeably increases the capacities of xPath by making it possible to

process and transform data. Moreover it is capable of handling data from different sources and to perform joints between the collected results.

The key components of the language are the "for", "let", "order by", "where" and return clauses, which respectively initiate loops, define variables, sort data, apply filter and return results of operations. xQueries also makes use of functions, which support recursion. These functions are declared thanks to the "define" operator.

Example

In the example below xQuery is used to collect data from two files (house-description.xml and house-prices.xml) and returns a single XML document.

```
<?xml version="1.0" encoding="UTF-8"?>
<houseprices>
{
for $x in doc("houses-description.xml)/houses/*
       for $y in doc("houses-prices.xml)/prices/*
              where $x/house_id = $y/house_id
              order by $y/price
              return <house>{ $x/house_id, $x/size, $y/price }</house>
 }
</houseprices>
```

The XML document returned would be in the format of the sample document listed below.

```
<?xml version="1.0" encoding="UTF-8"?>
<houseprices>
       <house>
              <house_id>1</house_id>
              <size>87</size>
              <price>72.000</price>
       </house>
       <house>
              <house_id>2</house_id>
              <size>62</size>
              <price>55.000</price>
       </house>
</houseprices>
```

WSDL

WSDL is a W3C recommendation. Current version is 2.0 and is split into three specification documents [WSDL20Primer] [WSDL20Core] and [WSDL20Adjuncts]. These documents have been published on June 26th, 2007.

WSDL stands for Web Service Description Language. A WSDL is an XML document used to describe and locate Web services.

A web service is described thanks to the four major components of the WSDL listed below.

- portType: this element contains the operations performed by the service. It can be compared to a function library.
- message: the messages used by the web service, and can be compared to the parameters and return values of a function. They can be composed of multiple parts
- type: the types of data used by a function.
- binding: the detail of protocol used to connect to the Web Service.

Example

In the example below the WSDL document describes the necessary component of a Web Service providing the price of elements identified by their "id" attribute.

Three message types are defined in the `<message>` blocks and two functions in the `<portType>` block. Access protocol and URI to these functions are provided in the `<binding>` block. In our case the protocol is SOAP through HTTP.

```
<message name="getPriceRequest">
	<part name="id" type="xs:string"/>
</message>

<message name="getPriceResponse">
	<part name="value" type="xs:string"/>
</message>

<message name="setPriceRequest">
	<part name="id" type="xs:string"/>
	<part name="price" type=”xs:integer”/>
</message>


<portType name="libraryPrices">
	<operation name="getPrice">
		<input message="getPriceRequest"/>
		<output message="getPriceResponse"/>
	</operation>
	<operation name="setPrice">
		<input message=”setPriceRequest”/>
	</operation>
</portType>

<binding type="libraryPrices" name="lP">
	<soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http" />
	<operation name="getPrice">
	<soap:operation soapAction="http://example.com/getPrice "/>
		<input>
			<soap:body use="literal"/>
		</input>
		<output>
			<soap:body use="literal"/>
		</output>
	</operation>
	<operation name="setPrice">
	<soap:operation soapAction="http://example.com/setPrices"/>
		<input>
			<soap:body use="literal"/>
		</input>
	</operation>

</binding>
```

SOAP

Introduction to SOAP

SOAP was originally defined by IBM and Microsoft. It is now a W3C recommendation and its current version is 1.2 second edition. It is split into three main specification documents [SOAP12Primer] [SOAP12Framework] and [SOAP12Adjuncts]. These documents have been published on April 27th, 2007.

SOAP (Simple Object Access Protocol) is a communication protocol used between applications. It lies at the application layer and relies on application transport protocols for communication. HTTP is the most common transport protocol for SOAP messages although other protocols such as SMTP are also valid.

SOAP messages use the XML language which ensures interoperability between systems, provides extensibility and leverages the widespread implementations found either in major software corporations and the open-source.

Structure of SOAP messages

A SOAP message must be constructed according to the four rules below.

- It must be encoded with XML;
- It must use standard namespace for encoding and the envelop (see below);
- It must not contain processing instructions.

SOAP messages are composed of four blocks:

- The envelop, which makes it possible to identify that the message is a SOAP message;
- The header, which contains generic information as well as data used to evaluate the capacity of the remote system to process the information;
- The body, which contains function calls, parameters and return values;
- The fault, with status and error code.

Example

The example below shows a SOAP request over HTTP. This request is made at the getPrice operation described in the preceding example.

```
 1 POST /getPrice HTTP/1.1
 2 Host: www.example.org
 3 Content-Type: application/soap+xml; charset=utf-8
 4 Content-Length: 309
 5
 6 <?xml version="1.0"?>
 7 <soap:Envelope
 8 xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
 9 soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
10  <soap:Body xmlns:m="http://www.example.org/prices">
11    <m:getPriceRequest>
12      <m:id >1001</m:id >
13    </m:getPriceRequest>
14  </soap:Body>
15 </soap:Envelope>
```

The request detailed above is an HTTP POST to the /getPrice URL. The SOAP part starts with the prologue on line 6, then the envelope which defines the document type of XML document (line 8) and the encoding style (line 9).

The "body" block (lines 10 to 14) contains the parameters used by the function call.

The response to this request is detailed below.

```
 1 HTTP/1.1 200 OK
 2 Content-Type: application/soap+xml; charset=utf-8
 3 Content-Length: 295
 4
 5 <?xml version="1.0"?>
 6 <soap:Envelope
 7 xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
 8 soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
 9  <soap:Body xmlns:m="http://www.example.org/prices">
10    <m:getPriceResponse>
11      <m:value>34.5</m:value>
12    </m:getPriceResponse>
13  </soap:Body>
14 </soap:Envelope>
```

This response contains the return value of the function on line 11.

Web Services components

Actors

Users

Users are the individuals indirectly requesting for a service through a specific user interface. The purpose of this interface is to hide the complexity of the application chain to the user. They are usually web pages displayed by a standard web browser. However there is no specific requirement for web-enabled interfaces.

Requesters

Requesters perform initiate the SOAP transaction with the web service. Its main tasks are to ensure that requests are properly formatted and that security measures are applied as expected by the provider.

A requester can be of three types:

- User controlled requester: these requesters are part of the software used by the user to request the service.
- Third-party requester: in this case the request to the web service is performed on behalf of the user who is not directly using a web service enabled software. This is the usual role of portals.
- Web Services: in many cases a Web Service needs additional data or processing provided by another one. In this case it requests the operation to the other Web Service. As a consequence it behaves like a requester.

Intermediary Web Services

Intermediary Web Services are part of the communication and processing chain between the requester and the provider. It may perform almost any tasks on the data in transit.

An XML firewall is a typical Intermediary Web Service as it receives the request, performs security checks, transforms some data and forward the request to the provider.

There may be multiple intermediary Web Services between the requester and the provider.

Web Service providers

The Web Service provider receives the request, processes it and provides an output to the requester. It communicates its requirements through WSDL.

It is responsible for setting the security parameters such as authentication or encryption.

Resources

Registries

Web Services discovery is made possible through a standard called UDDI for Universal Description, Discovery and Integration. This standard facilitates the access to Web Services in providing description and access point.

Access points are provided in the form of URL pointing to a WSDL file which, in turns, details functions available to Web Service requesters.

Portal

The portal is the common interface between the user and the Web Service. They initiate transaction with web services on behalf of the user. Consequently the user has no visibility or knowledge of the Web Services involved in the processing of its data or request.

Communications

All communications between the components of a Web Service infrastructure are based on SOAP messages, would it be between a requester and a registry server, a requester and an intermediary Web Service, a requester and a provider etc.

Coordination

The need for coordination

When multiple actors are involved in a Web Service infrastructure, it becomes necessary to organize the processing of data and the communication chain between each of them. This is called coordination and can be done in two different ways:

- Orchestration: for Web Services relying on actors belonging to the same entity (company, administration etc.) and responding to the same authority;
- Choreography: used when Web Services actors are split in different entity and depends on different authorities.

Orchestration

Orchestration is implemented from a primary Web Service which will rely on multiple others to provide requesters with the response they expect. This primary Web Service is called the encapsulating Web Service as it behaves like a frame containing parts of data retrieved thanks to other Web Services.

An orchestration typical architecture is described below.

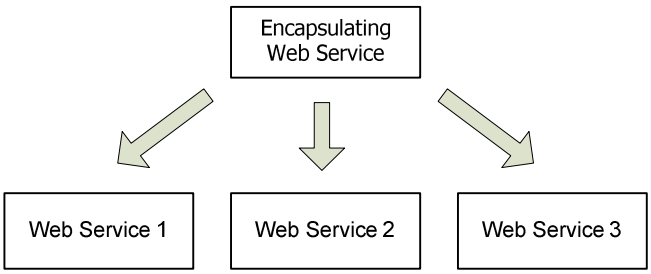

Figure 1: Web Services Orchestration

Choreography

In this case there is no Web Service controlling the process as Web Service are split between independent entities. Therefore choreography is mainly about describing the relationship between Web Services so that they are capable of communicating with each other in order to perform a process.

An example of orchestration architecture is provided in the figure below.

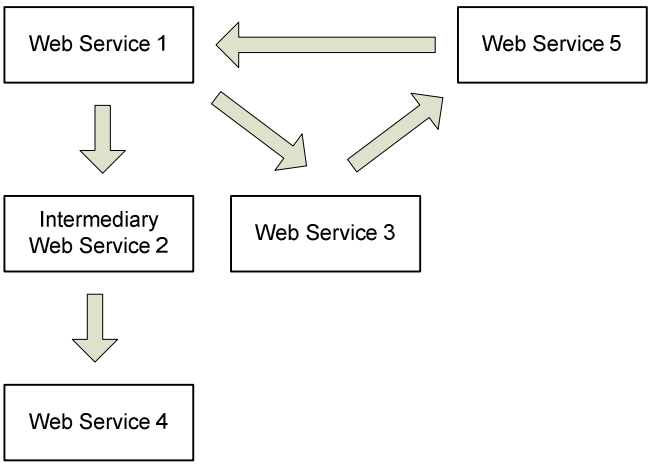

Figure 2: Web Services Choreography

Standards for Web Service security

Bricks of web services security

Scope of security

Standards have been designed in order to provide security at different layer of the web service:

- Messaging: as SOAP was not designed with security in mind, additional protection is needed to secure transactions between the components of a Web Service infrastructure, either from a privacy or an availability point of view;
- Access to resources: services and resources provided by Web Services are not necessarily intended for use by everybody. Therefore authentication and authorization mechanisms have to be setup and maintained across components involved in the delivery of the service;
- Negotiation between parties: Web Services are usually built thanks to multiple parties, involved in the processing of different parts of the request or in chain. Automation of service discovery and usage makes it necessary to negotiate a contract of use, would it be technical, business-oriented or both.
- Trust relationship: transmission of data to third-parties, which in turn may make use of third-parties for additional processing requires the establishment of trust relationship between involved parties as one may have to trust services or users he doesn't have control upon.

Standards overview

Security layers

Main available standards, would they be specially designed for web services or not are presented in the figure below, along with the network or application layer they apply to.

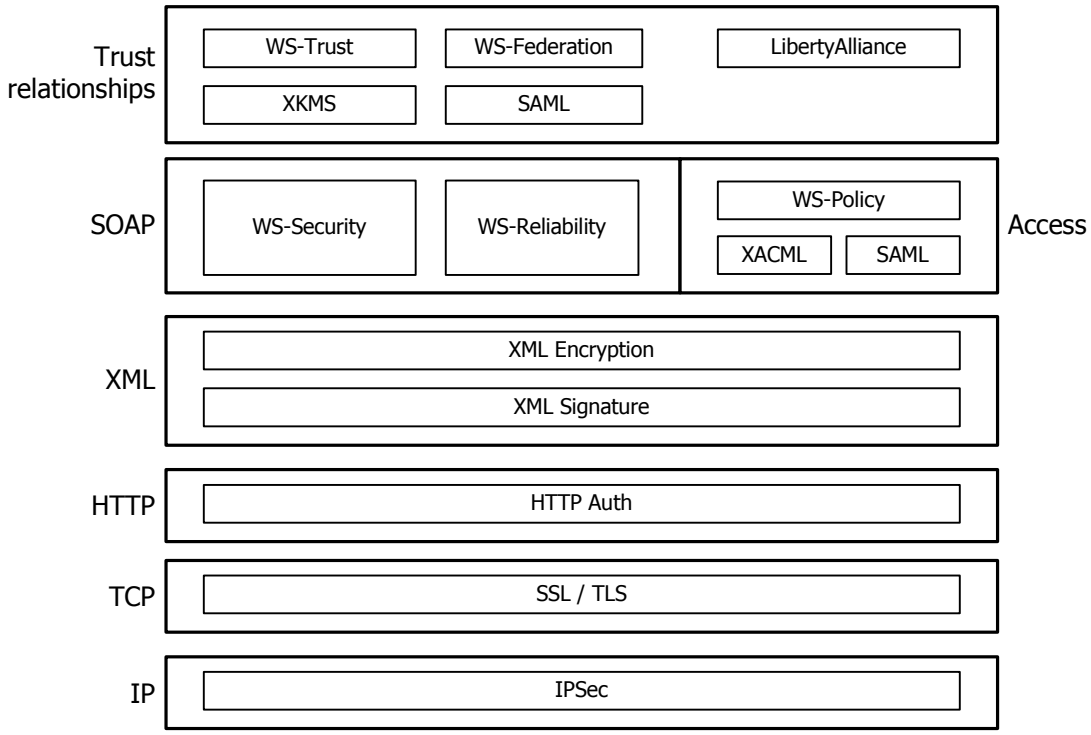

Figure 3: Web Services Security Standards

Network and transport layers

Network and transport layers security is nothing special in regard to Web Services. Available technologies and their current implementations are mainly aiming at ensuring point-to-point confidentiality and integrity of data in transit.

While IPSec implementations remain quite limited in scope, SSL and TLS are very common and can be found in almost every website.

Moreover those protocols can also ensure, to a certain extent, authentication of both endpoints via the use of certificates. This is typically the case of SSLv3 and TLS which are sometimes used for such purpose.

Application transport layer

The only form of security provided at the application transport layer is HTTP authentication, which can be implemented by two different mechanisms: Basic and Digest.

While the Basic authentication security level is particularly low, the Digest one provides some additional security mechanisms which make it more difficult to crack.

Whatever mechanism is used they are both considered more or less insecure and are usually implemented over secured network or transport layer security mechanism which ensures stronger encryption and protection against replay attacks.

Presentation layer

Two security mechanisms are available at the presentation layer: XML Encryption [XMLENC] and XML Signature [XMLDSIG], which are both W3C standards. They respectively provide the capability to encrypt and sign XML data.

The main difference with similar mechanisms implemented at the network, transport and application layers is the capability to sign and/or encrypt only parts of the message. As service delivery to end user may involve several actors from potentially entities, the capability to ensure the confidentiality and the integrity of specific parts of the transmitted data is mandatory in many cases.

Another necessary mechanism is the support for specific representations as encoding may change on the path, thus invalidating signatures. As a consequence XML Signature supports two types of canonization methods, providing most of the functions which would be necessary to normalize the signed data before the signature is checked.

SOAP Layer

Most of current security implementations rely on the protection of SOAP messages exchange. Indeed as we are still in the early stages of secured web services deployment, the level of security provided by the standards at this layer is an acceptable compromise between maturity of available technology, structural impacts on existing infrastructures and protection against main threats.

Two major standards are made available for message security: WS-Security [WSSE11] and WS-Reliability [WSRM11].

The latest mainly aims at ensuring quality and availability of SOAP communications. On the other hand WS-Security is a subset of XML Signature and XML Encryption together with additional mechanisms which ensure the transmission of authentication tokens along the path and provide anti-replay protection.

Authorizations and Policies

Authorizations are managed with a set of three components: SAML, XACML and WS-Policy.

SAML

SAML is an OASIS[2] standard published on March 15th, 2005. It is currently in version 2.0 [SAML20].

SAML stands for Security Assertion Markup Language. It is an XML dictionary which defines a framework to exchange security assertions between actors of a web service.

SAML involves two parties: the assertion party which provides information regarding a subject (such as a user, its authentication status and its attributes), and the relying party which chose or not to trust the assertion and makes decision according to the information read in the assertion.

XACML

XACML is an OASIS standard. Current version is 2.0 [XACML20] and has been published on February 1st, 2005.

XACML is an XML-based language used to represent and enforce security policies. Therefore it defines both a policy language and a communication language with request and responses. This communication is necessary to know whether or not a particular action is authorized or forbidden.

XACML and SAML interaction can be described as follow: the XACML will grant or deny access to a resource, then the appropriate SAML assertion is created and processed accordingly by relevant actors of the web service.

WS-Policy

WS-Policy is a W3C recommendation currently in version 1.5 [WSP15] which has been published on September 4th, 2007

WS-Policy is a global specification which makes it possible to define the necessary requirements to communicate with a Web Service. It covers three domains: security, reliable messaging and addressing.

Security policies are specified in the WS-SecurityPolicy [WSSP12] standard where several assertions are defined regarding integrity, confidentiality and security tokens.

Trust relationship

The inherent distributed nature of Web Services makes it mandatory to implement a mechanism of trust between parties.

When a small amount of actors responding to the same authority are involved, basic mechanisms can be implemented. This is the case of pairwise trust cycles, where each actor shares its public key with each other.

However this kind of architecture is not applicable to larger infrastructures. Federations have been designed to handle such cases. They involve a Trusted Third-Party (TTP), which provides necessary keys on-demand. When several organizations are involved a pairwise trust cycle is established between TTP of each organization.

Two standards are currently available and provide the necessary framework to define and implement federations of trust: Liberty Alliance on one Hand, and WS-Trust [WST13] and WS-Federation [WSFL1.1] on the other hand.

WS-Trust relies on WS-SecurityPolicy to determine the tokens which are required to communicate between web services. WS-Federation defines trust realms across which Web Service are able to communicate. It specifies the protocols and the way Web Service requesters should interact with Web Service providers and Security Token Services.

[2] OASIS - Organization for the Advancement of Structured Information Standards - http://www.oasis-open.org/

XML Security

XML Signature

XML Signature is a W3C recommendation. The current document is the second edition [XMLDSIG] and has been published on June 10th, 2008.

Signature overview

General concerns regarding XML Signature

The main purpose of XML signature is to provide a mechanism which makes it possible to sign only specific parts of a document. As XML documents are made of multiple blocks which may be authored by different persons or systems, the need for ensuring the origin and the integrity of blocks seems legitimate.

According to the position of the signature in the document we can distinguish three cases:

- Enveloping signature: when the data signed are part of the signature itself;
- Enveloped signature: when the signature is applied to data of the same document;
- Detached signature: when the signature is used to sign data which are located outside the current document.

The first step to create signature is to generate a digest for each block to be signed (called reference). All references are then included into a group of elements. This group of elements is then digested and digitally signed.

Consequently an XML signature is made of one or more references

Structure of XML Signatures

The generic structure of XML Signatures is given below.

```
<Signature ID?>
     <SignedInfo>
       <CanonicalizationMethod/>
       <SignatureMethod/>
       (<Reference URI? >
         (<Transforms>)?
         <DigestMethod>
         <DigestValue>
       </Reference>)+
     </SignedInfo>
     <SignatureValue>
    (<KeyInfo>)?
    (<Object ID?>)*
</Signature>
```

Main components are:

- SignedInfo is the very element which is signed. It can contain digests for several parts of the document;
- CanonicalizationMethod: is a reference to the mechanism used to normalize the representation of the signed data;
- SignatureMethod: is the algorithm used to sign the content of the `<SignedInfo>` block;
- Reference: is a pointer to the signed resource;
- Transforms: is an ordered list of operations performed on the data prior to its signing;
- DigestMethod: is the algorithm used to generate the digest of the data to be signed, after the transforms have been applied;

- KeyInfo: identifies the key needed to validate the signature
- Object: are external blocks which make it possible to include additional data within the signature element. A typical element is the `<SignatureProperty>` block which includes a timestamp of the message.

The figure below show the structure of the block.

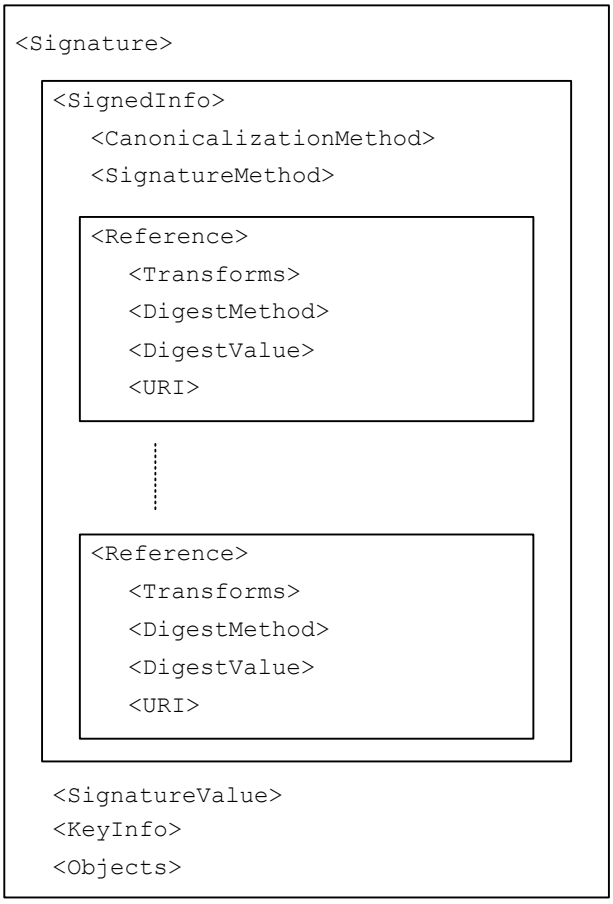

Figure 4: The <Signature> block

Core operations

Core operations perform signature validation and integrity checking of XML elements.

Signature generation

Two steps are to be taken in order to generate a signature: reference generation which is used for integrity checking of each block to be signed and signature generation for the group of references.

- Reference Generation
    1. Transformations are applied in the order they are listed in the `<Transforms>` block;
    2. The digest is calculated on the result of the transform operations;
    3. The `<Reference>` block is created.
- Signature Generation
    1. The `<SignedInfo>` block is created with the `<CanonicalizationMethod>`, `<SignatureMethod>` and all the `<Reference>` blocks ;

2. The block is canonized according to the method specified in the <CanonicalizationMethod> block;
3. The block is signed according to the method specified in the <SignatureMethod> block;
4. The <Signature> block is constructed with the necessary block: <SignedInfo>, <Object>, <KeyInfo> and <SignatureValue>.

Signature Validation

As for signature generation, signature validation is performed in two steps: reference validation which intends to ensure the content hasn't been modified and signature validation which checks if the signature is valid.

- Reference validation
    1. The <SignedInfo> block is canonized according to the method specified in <CanonicalizationMethod>;
    2. The digest for data object of each <Reference> block is calculated;
    3. The result is compared to the <DigestValue> of the <Reference> block.
- Signature validation
    1. The <SignatureMethod> is canonized according to the method specified in the <CanonicalizationMethod> block;
    2. The result is used together with the <KeyInfo> to validate the signature of the <SignedInfo> block.

Canonicalization

XML Canonicalization

The purpose of canonization is to provide a standard representation of a subset of a document, without any consideration regarding the context. The goal is to provide a mechanism that would lead to the uniqueness of this representation.

Such a mechanism is mandatory in the case of signature and integrity checking of XML data exchanged between the actors of a Web Service. Indeed the heterogeneous nature of components and the openness of XML makes it possible to apply transformations which would not impact the interoperability between systems (and therefore satisfy the main objective of the web services infrastructure), but would confuse validation mechanisms.

Canonicalization of XML documents is based on the processing of data represented according to the xPath data model. The node-set to be canonized is the first parameter to provided. A second one is a simple flag which indicates if the comments should be included in the canonical form or not. Canonicalization is performed according to the standards XML canonical methods. As of today two canonical forms exists for XML 1.0 and 1.1 documents: inclusive [XMLC14N] and exclusive [XMLC14N-EXC] canonicalization.

Inclusive and exclusive canonicalization

The difference between the exclusive and inclusive canonicalization methods is the handling of namespaces and xml attributes inherited from ancestors.

The default canonicalization method (later called "inclusive"), specifies for elements which have no ancestors in the node-set (apex nodes), that all the ancestor elements are to be analysed and their attributes added to a list of attributes. From this list are removes attributes which can be found in the descendants of the apex node. Then the list is alphabetically merged with the list of attributes of the apex node.

The inclusive canonicalization method is not appropriate to normalize parts of documents included in an envelop which could change along the processing by different systems on the path, as shown in the example below.

Example

In the example below we generate the canonical form of the element elem1 whose envelop changes.

Document	Canonical form of elem1
<n0:pdu xmlns:n0="http://a.example"> <n1:elem1 xmlns:n1="http://b.example"> content </n1:elem1> </n0:pdu>	<n1:elem1 xmlns:n0="http://a.example" xmlns:n1="http://b.example"> content </n1:elem1>
<n2:local xmlns:n2="http://c.example"> <n1:elem1 xmlns:n1="http://b.example"> content </n1:elem1> </n0:pdu>	<n1:elem1 xmlns:n1="http://b.example" xmlns:n2="http://c.example"> content </n1:elem1>

Table 2: Inclusive canonicalization limitation

In this case the two canonical forms are different.

As a consequence the exclusive canonicalization method has been specified. The major difference with the inclusive canonicalization method is that the search and analysis of ancestors' attributes is not performed. Therefore changes in the envelop of the element to be canonized will not impact the canonical form of the element itself.

Formatting rules

Rules are applied according to the nature of the node to be canonized: root, element, attribute, namespace, text, processing instruction and comments.

Main rules which are applied are:

- Canonical form of the document is UTF-8 encoded;
- Line breaks are transformed to #xA;
- Attribute values are normalized;
- Attribute value delimiters are set to ";
- Entity references are replaced;
- Character references are replaced in text node (&#x32 is transformed to 2);
- CDATA sections are replaced by their character content;
- XML declarations and Document Type Declaration (DTD) are removed;
- Empty elements are replaced by start-end tag pair (<data/> becomes <data></data>);
- Whitespaces are preserved except for node which do not belong to the node-set;
- Whitespaces are normalized into tags, only one space between parameters of the tag;
- Replacement in text nodes of &,<, >, " and #xD respectively by & < > " and &#xD

Limitations of the canonicalization

Two XML documents can be different but logically equivalent, according to the application context and processing. The usual example is the possible equivalence (in the context of a specific application) of the expressions `<color>black</color>` and `<color>rgb(0,0,0)</color>`. There is currently no way to circumvent this limitation.

Other limitations are related to the loss of specific information which are not available in the data model. The main issue is the loss of the base URI, which is used to access relative URI. Documents which contain this type of URI are not appropriate for canonicalization.

The <Signature> element

<Signature> block overview

The <Signature> block contains all the information necessary to validate the signed parts of the XML document. It is composed of 2 mandatory components and 2 optional ones, respectively the signature info, the signature value, the key info and the optional object component.

Signature generation

The way signatures are generated is specified in the `<SignatureMethod>` element of the signature info block `<SignatureInfo>`. They can be generated either through a MAC (Message Authentication Code) or a Digital signature algorithm. The list below provides the recommendation in terms of algorithm implementation. However the model is voluntarily made open so that alternative mechanisms can be specified and implemented.

- MAC algorithm:
    - HMAC-SHA1: REQUIRED
- Digital signature algorithms:
    - DSA: REQUIRED
    - PKCS1 (RSA-SHA1): RECOMMENDED

The signature value is stored in the `<SignatureValue>` element and is the Base64 encoding of the signature function output.

Signature keys

As interoperability must be guaranteed between the components of the web service infrastructure, it is necessary to provide an open and scalable way to transmit signature keys information. This is achieved thanks to the `<KeyInfo>` element which contains one of several types of elements:

- Key name: a generic text field which is used to communicate a key identifier to the recipient;
- Key value: the key to be used in order to validate the signature. Two specific formats are defined for DSA and RSA keys;
- Retrieval method: specifies a way to use a `<KeyInfo>` block from a reference either external or local to the document;
- X509 data: an identifier of keys or x509 certificates;
- Key data: is a specific element used to transmit data regarding signature keys needed for two specific mechanism: PGP and SKIP.

It is also possible that the key information is shared by the actors of the communication or transmitted via other mechanisms. Therefore the `<KeyInfo>` block is optional.

References

References are identifiers of document blocks which have been signed. The `<Reference>` element may occur multiple times in the document. The main components of the `<Reference>` element are:

- A URI reference, which points to an element internal or external to the document;
- A digest algorithm, used to generate the hash of the element to be signed;
- A digest result, which is the output of the digest algorithm applied to the referenced element.

Multiple references can be held into the `<SignedInfo>` block, thus making it possible to generate a single signature for several blocks split over the XML document.

The support of SHA-1 as a digest algorithm is required by the standard, while MD5 which is suspected of several weaknesses is explicitly not recommended.

XML Encryption

XML Encryption is a W3C recommendation. The current document [XMLENC] was published on December 10th, 2002.

XML Encryption overview

Encryption Granularity

XML encryption has been designed to make it possible to encrypt whole or parts of an XML document. This is made necessary as intermediate Web Services may not be authorized to access some of the contents either send by the requester or by the service provider.

As a consequence XML encryption can be applied:

- To the totality of the document;
- To an XML element;
- To the content of an XML element;
- To the data part of an XML element.

Example

The table below show different scopes of encryption of a simple XML document.

Scope	Document

No Encryption	`<?xml version='1.0'?>`
	`<PaymentInfo xmlns='http://example.org/payment'>`
	`  <Name>John Smith</Name>`
	`  <CreditCard Limit='5,000' Currency='USD'>`
	`  <Number>4019 2445 0277 5567</Number>`
	`  <Issuer>Example Bank</Issuer>`
	`  <Expiration>04/02</Expiration>`
	`  </CreditCard>`
	`</PaymentInfo>`

Document Encryption	`<?xml version='1.0'?>`
	`<EncryptedData xmlns='http://www.w3.org/2001/04/xmlenc#'`
	`MimeType='text/xml'>`
	`<CipherData>`
	`<CipherValue>A23B45C56</CipherValue>`
	`</CipherData>`
	`</EncryptedData>`

Element Encryption	`<?xml version='1.0'?>`
	`<PaymentInfo xmlns='http://example.org/paymentv2'>`
	`<Name>John Smith</Name>`
	`<EncryptedData Type='http://www.w3.org/2001/04/xmlenc#Element'`
	`xmlns='http://www.w3.org/2001/04/xmlenc#'>`
	`<CipherData>`
	`<CipherValue>A23B45C56</CipherValue>`
	`</CipherData>`
	`</EncryptedData>`
	`</PaymentInfo>`

Element Content Encryption	`<?xml version='1.0'?>`
	`<PaymentInfo xmlns='http://example.org/paymentv2'>`
	`<Name>John Smith</Name>`
	`<CreditCard Limit='5,000' Currency='USD'>`
	`<EncryptedData xmlns='http://www.w3.org/2001/04/xmlenc#'`
	`Type='http://www.w3.org/2001/04/xmlenc#Content'>`
	`<CipherData>`
	`<CipherValue>A23B45C56</CipherValue>`
	`</CipherData>`
	`</EncryptedData>`
	`</CreditCard>`
	`</PaymentInfo>`

Element Data Encryption	<?xml version='1.0'?> <PaymentInfo xmlns='http://example.org/paymentv2'> <Name>John Smith</Name> <CreditCard Limit='5,000' Currency='USD'> <Number> <EncryptedData xmlns='http://www.w3.org/2001/04/xmlenc#' Type='http://www.w3.org/2001/04/xmlenc#Content'> <CipherData> <CipherValue>A23B45C56</CipherValue> </CipherData> </EncryptedData> </Number> <Issuer>Example Bank</Issuer> <Expiration>04/02</Expiration> </CreditCard> </PaymentInfo>

Table 3: Scope of XML encryption

Structure of XML encrypted blocks

The generic structure of the XML encrypted blocks is defined as expressed below.

```
<EncryptedData Id? Type? MimeType? Encoding?>
    <EncryptionMethod/>?
    <ds:KeyInfo>
      <EncryptedKey>?
      <AgreementMethod>?
      <ds:KeyName>?
      <ds:RetrievalMethod>?
      <ds:*>?
    </ds:KeyInfo>?
    <CipherData>
      <CipherValue>?
      <CipherReference URI?>?
    </CipherData>
    <EncryptionProperties>?
</EncryptedData>
```

The main components of the structure are:

- EncryptionMethod: which detail the algorithm used for encryption;
- KeyInfo: contains the necessary data regarding the encryption key, the way to retrieve it or to calculate it;
- CipherData: contains the encrypted data or reference to this data.

Core Operations

Encryption

Steps for encryption are described below:

1. Encryption algorithm is chosen;
2. Construct the `<KeyInfo>` block
    a. With the name, a reference or the key itself;
    b. With the key, encrypted with the same mechanism as currently described.
3. Serialize the data
    a. To UTF-8 if the data is an XML element or element content;
    b. To octets if the data are of any other type that is not octets.

4. Encrypt the data with the selected algorithm and key;
5. Build the appropriate structure to store the encrypted data information (<EncryptedData> or <EncryptedKey>) and provide encrypted data storage location information, either a <CipherData> block for local storage or an URI reference for external storage.

Decryption

1. Encryption algorithm and parameters as well as <KeyInfo> data are retrieved;
2. Encryption key is located as well as the key used to decrypt it if necessary;
3. Locate the encrypted data/key, either in the document or from external source if a reference URI is provided;
4. Decrypt the data/key with the algorithm and key provided in steps 1 and 2.

Encrypted data

<EncryptedType>
All encrypted data or keys elements are constructed on the same abstract type: <EncryptedType>.

This type defines several blocks which can be grouped in two categories :

- Encryption material: a group which contains information about the encryption method, the keying material, the encrypted data itself and additional encryption properties;
- Data information: to which belongs optional information such as an ID of the block, the XML type and MIME type of the encrypted data and its encoding, if any.

Encryption Method

Any encryption method can be used to generate ciphers. However some are explicitly required or recommended and are provided with a reference URI.

Algorithm	Status	URI
Triple-DES	REQUIRED	http://www.w3.org/2001/04/xmlenc#tripledes-cbc
AES 128 bits	REQUIRED	http://www.w3.org/2001/04/xmlenc#aes128-cbc
AES 192 bits	OPTIONAL	http://www.w3.org/2001/04/xmlenc#aes192-cbc
AES 256 bits	REQUIRED	http://www.w3.org/2001/04/xmlenc#aes256-cbc

Table 4: Encryption Algorithm

The URI specified in the table above is an attribute of the <EncryptionMethod> element.

Cipher data and references

Encrypted data can be directly included into the <EncryptedType> derived element. In this case it is contained into a <CipherValue> element.

It is also possible that the <CipherValue> is not directly supplied. Then a <CipherReference> element identifies a source from which the encrypted data can be retrieved. Additionally, transforms operations can be specified as encrypted data could have been transformed prior to its storage into the external document.

Example

In the example below the encrypted data is retrieved from the URI http://www.example.com/CipherValues.xml (line 0) and extracted with the xPath expression specified on line 5. Then the retrieved value is Base64 decoded as stated in the <Transform> element on line 8.

```
0 <CipherReference URI="http://www.example.com/CipherValues.xml">
1     <Transforms>
2       <ds:Transform
3      Algorithm="http://www.w3.org/TR/1999/REC-xpath-19991116">
4           <ds:XPath xmlns:rep="http://www.example.org/repository">
```

```
 5             self::text()[parent::rep:CipherValue[@Id="example1"]]
 6           </ds:XPath>
 7       </ds:Transform>
 8       <ds:Transform Algorithm="http://www.w3.org/2000/09/xmldsig#base64"/>
 9     </Transforms>
10   </CipherReference>
```

Encryption key components

It is optional to provide the encryption key into the <EncryptedData> block as both sides of the communication may exchange key information through an out-of-band mechanism. As a consequence the <ds:KeyInfo> block is optional.

Keying material

The necessary elements needed to decrypt the data or the encryption key can be provided by an element of the <ds:KeyInfo> block. In this case the information provided can be:

- The key value, stored in the <ds:KeyValue> block;
- The key name, a referrer to the <CarriedKeyName> element of an <EncryptedKey> block;
- A reference to the key location, expressed as an URI.

In the case of an <EcryptedKey> block located outside the <ds:KeyInfo> block, it is possible to specify the <EncryptedData> or <EncryptedKey> block to which the key shall be applied.

Last, it is possible that all materials are handled by applications and therefore there will be no need to have them transported into the XML document.

<EncryptedKey> element

The <EncryptedKey> element provides three extensions to the <EncryptedType> abstract type:

- <ReferenceList>: this element is optional and contains pointers to the data and keys encrypted with this key. Several data and keys can be referenced into this element.
- <CarriedKeyName>: this element is also optional and provides a human-readable description of the key contained into the <EncryptedKey> element. When the <CarriedKeyName> element is defined, its value can be used to reference the key from a <ds:KeyInfo> element.
- <Recipient>: is an optional attribute to the <EncryptedKey> element. It provides information about the recipient of the key. Its value will depend on the application and no standard is defined.

Encrypted data and encrypted key relationship

Example

In the example below an <EncryptedData> element (with ID 'ED') is defined which uses a key defined in an <EncryptedKey> element (with ID 'EK') located outside the <ds:KeyInfo> element.

```
 1 <EncryptedData Id='ED'
         xmlns='http://www.w3.org/2001/04/xmlenc#'>
 2   <EncryptionMethod
           Algorithm='http://www.w3.org/2001/04/xmlenc#aes128-cbc'/>
 3   <ds:KeyInfo xmlns:ds='http://www.w3.org/2000/09/xmldsig#'>
 4     <ds:RetrievalMethod URI='#EK'
             Type="http://www.w3.org/2001/04/xmlenc#EncryptedKey"/>
 5     <ds:KeyName>Julie MARTIN</ds:KeyName>
 6   </ds:KeyInfo>
 7   <CipherData><CipherValue>DEADBEEF</CipherValue></CipherData>
 8 </EncryptedData>
 9 <EncryptedKey Id='EK' xmlns='http://www.w3.org/2001/04/xmlenc#'>
10   <EncryptionMethod
           Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
```

```
11   <ds:KeyInfo xmlns:ds='http://www.w3.org/2000/09/xmldsig#'>
12     <ds:KeyName>Marcel DUPOND</ds:KeyName>
13   </ds:KeyInfo>
14   <CipherData><CipherValue>xyzabc</CipherValue></CipherData>
15   <ReferenceList>
16     <DataReference URI='#ED'/>
17   </ReferenceList>
18   <CarriedKeyName>Julie MARTIN</CarriedKeyName>
19 </EncryptedKey>
```

On line 2 the chosen encryption algorithm is AES 128 bits.

On line 4 the `<ds:KeyInfo>` element contains a `<RetievalMethod>` element pointing to an URI referencing the element of the document with ID 'EK'.

On line 5 the key is identified by its name with the value 'Julie MARTIN'.

On line 7 is the encrypted data.

On line 9 the `<EncryptedKey>` element with ID 'EK' is defined.

On line 16 the ID ('ED') of the referencing URI is specified.

On line 18 the name of the key carried by the `<EncryptedKey>` element is expressed. It matches the value of the `<KeyName>` element defined on line 5.

SOAP messages security: WS-Security

WS-Security is an OASIS standard, currently in version 1.1 [WSSE11]. It was published on February 1st, 2006.

Purpose and scope of WS-Security

WS-Security aims at providing the capability to implement an end-to-end security mechanism which would be independent of the transport protocol. This goal is achieved by providing security methods to SOAP through the use of standard extensions into the header.

These extensions are based on the components listed below:

- XML encryption: used to encrypt parts of the message and critical components of the header;
- XML signature: which makes it possible to sign and ensure integrity of the message and its headers;
- Security tokens: which transport "claims" to appropriate recipients of the message.

The openness of technologies implemented in WS-Security and their use at the message level ensures that different parts of the message can be protected in different ways. Moreover messages can be considered as self-protected as their security is not dependant of the transport level.

However, WS-Security does not provide comprehensive security against all possible threats to a SOAP message. Technologies applied into WS-Security ensure message confidentiality through encryption, guarantee of origin and integrity through digital signing and transport of authentication credentials through security tokens.

On the other hand, and as stated in the specifications, WS-Security by itself does not provide security against some attacks, such as replay attacks, use of weak passwords, man-in-the-middle attacks or lack of protection of security tokens.

Moreover, the potential complexity of large and distributed application chains using WS-Security may introduce weaknesses which could not be easily identified, particularly in the case of infrastructure involving third-parties belonging to external organizations.

The Security Header

WS-Security defines the use of a new header block `<wsse:Security>` which contains all the necessary elements to process secured information transported by the SOAP message.

Such header is targeted to specific recipients. As a consequence multiple `<wsse:Security>` headers may be present in a SOAP messages, as long as each of them targets a different recipient.

On the other hand, if an active intermediary needs to apply additional security measures to be handled by a target referenced in an existing `<wsse:Security>` header, the intermediary must add a sub-element to this header. WS-Security does not enforce the ordering of sub-element, however, it is recommended that sub-element should be prepended to the previous ones. As a consequence it is expected that the receiver will be able to process sub-elements in the order they have been added to the `<wsse:Security>` header block.

Logically it is also recommended that a sub-element containing a key should precede the element using the key.

The attributes of the WS-Security header are listed in the table below.

Attribute	Description	Comments
S11:actor	Identifies a SOAP 1.1 actor	Optional. Only one <wsse:Security> block can omit this attribute. No two instances of <wsse:Security> block can have the same S11:actor attribute
S12:role	Identifies a SOAP 1.2 role	Optional. Only one <wsse:Security> block can omit this attribute. No two instances of <wsse:Security> block can have the same S12:role attribute
S11:mustUnderstand	A SOAP 1.1 attribute which indicates if processing of the header is mandatory or not.	Possible values are 0 and 1. Default is 0.
S12:mustUnderstand	A SOAP 1.2 attribute which indicates if processing of the header is mandatory or not.	Possible values are 0 and 1. Default is 0.

Table 5: <wsse:Security> attributes

XML encryption and XML signature in WS-Security

Although WS-Security relies on the XML encryption and XML signature standards, some particularities and limitations are recommended for the implementation of WS-Security. Unfortunately these recommendations are presented through the use of keywords such as SOULD or SHOULD NOT, then letting the possibility to implement mechanisms which may expose to loss of interoperability or potential security weaknesses.

XML Signature

Support of XML signature is mandatory for the compliance with WS-Security.

The main recommendation of WS-Security regarding signatures takes into account the fact that a SOAP message will be enriched with multiple components along the way to its destination. It means that parts which are signed may be included into elements which will also be signed by another party and so on. This raises two constraints, illustrated by the case below.

A message is created by a first Web Service (WS1) with a header and a body. These parts are to be signed by WS1. A second one (WS2) adds new header and body, which contains previous headers and body. If WS1 and WS2 whish to sign their respective parts (then WS2 signing also the elements provided by WS1), it is necessary that:

- WS2 can access the elements provided by WS1, which is not made possible if WS1 implements enveloping signature (the signed element is then contained into a `<Object>` element of the `<Signature>`). As a consequence, enveloping signature should not be used;
- The element accessed by WS2 can be signed as-is, and therefore no transformation (an particularly the Enveloped signature transform) should be needed prior to generating the signature. Then enveloped signature transform should not be used.

A last constraint, which makes sense in the context of WS-Security, is that the content to be signed should be included into the message. This is logical as WS-Security is about signing SOAP messages, and therefore should not be applied to external content.

XML Encryption

Support of XML encryption is mandatory for the compliance with WS-Security.

WS-Security implements the XML Encryption recommendation in order to perform encryption of any combination of header and body blocks or any of their sub-elements. This flexibility is obtained by leveraging some of the XML Encryption functions and adding an additional element: `<wsse11:EncryptedHeader>`.

Encrypted data are contained into `<EncryptedData>` elements in the SOAP envelop, except for encrypted SOAP headers which are found in the new `<wsse11:EncryptedHeader>` element.

In order to identify which parts of the SOAP message are encrypted, encrypted elements are to be referenced by `<DataReference>` elements into the `<ReferenceList>` element from the XML Encryption. This mechanism is an extension of the XML encryption definition of the `<ReferenceList>` element, which was originally designed to be located into an `<EncryptedKey>` element and was used to reference blocks to be decrypted with the same key.

However, WS-Security considers that it is possible that elements of the reference list would be decrypted by different keys. In this case, keys must be located into specific `<KeyInfo>` blocks.

Security tokens

The purpose of security tokens specified in WS-Security is to transport and sign "claims" to the target of the message. However, WS-Security does not specify or enforce the way claim confirmation should be done. As a consequence tokens can be used in various ways, depending on the implementation of the web service and the level of interoperability expected.

Three types of tokens are defined: username, binary and XML tokens. They can be used either to transmit authentication credential, proof of identity, session information etc.

A usual scenario of token utilization is described in the figure below.

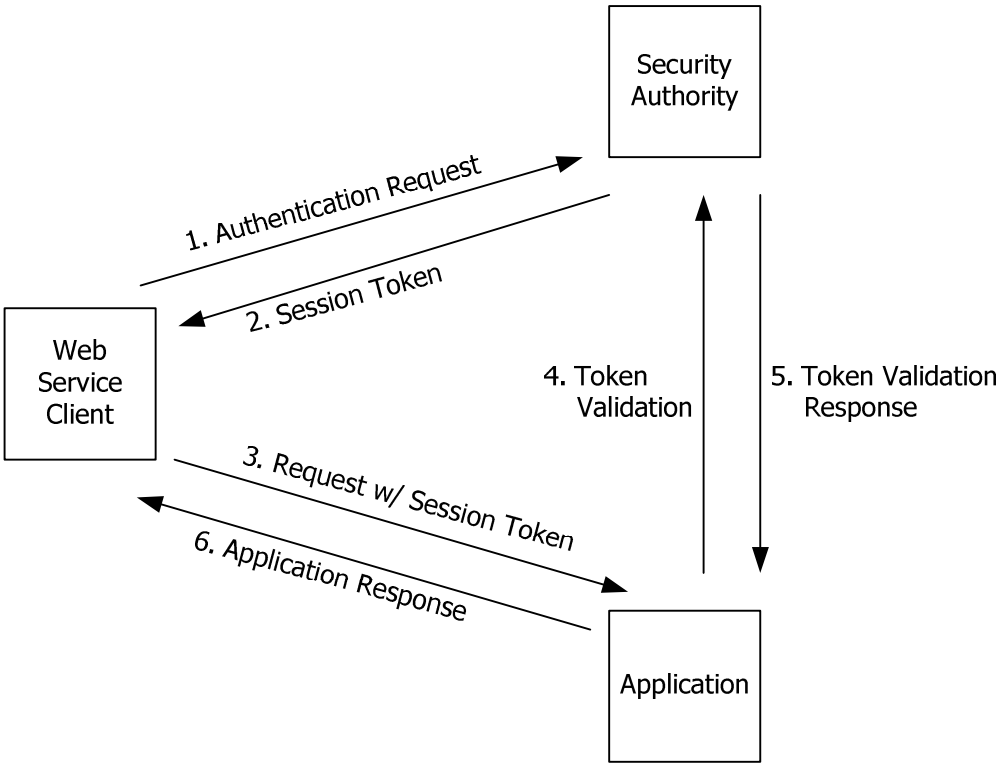

Figure 5: Security Token usage scenario

Username Tokens

These are the simplest type of token which can be used and are logically used to provide a username. The token is identified by a <wsse:UsernameToken> element which contains the username into a <wsse:Username> element.

In the case where the authentication is performed by the application itself, a password can be specified into the token, either in cleartext and through a digest. The first one is not recommended for obvious security reasons, unless it is used to transmit one-time password such as S/Key password.

The digest password are created together with a nonce and a timestamp (<Created> element). The digest is then calculated according to the formula below:

Password_Digest = Base64 (SHA-1 (nonce + created + password))

Example

Below are three examples of <wsse:UsernameToken> elements: a simple one containing only the username, one with a cleartext password and one with a password digest.

Simple token

```
<wsse:UsernameToken>
       <wsse:Username>Paul Smith</wsse:Username>
</wsse:UsernameToken>
```

ClearText password token

```
<UsernameToken>
       <Username>Paul Smith</Username>
       <Password Type="PasswordText">MyPassword</Password>
       <Nonce>123521</Nonce>
       <Created>2005-11-24T15:00:00Z</Created>
</UsernameToken>
```

Digest password token

```
<UsernameToken>
       <Username>Paul Smith</Username>
       <Password Type="PasswordDigest">XYZAAA9</Password>
       <Nonce>123521</Nonce>
       <Created>2005-11-24T15:00:00Z</Created>
</UsernameToken>
```

Binary and XML Tokens

The type "XML Tokens" mainly refers to SAML assertions while any XML-based token (such as XrML – eXtensible Rights Markup Language, or XCBF – XML Common Biometric Format) is theoretically supported into this type of elements. These elements are usually signed and self-verifying, making the use of a validation third party unnecessary.

In this case the token usage scenario is simplified, as there is no need for the application to get credentials / token validation by the Security Authority.

WS-Security also implements a `<wsse:BinarySecurityToken>` element in order to enable the generic use of non-XML authentication tokens. These security tokens are referenced as "Signed Security Tokens" as they are asserted and cryptographically signed by a specific authority. Two types of binary tokens are explicitly named into the WS-Security standard: X.509 Certificates and Kerberos Tickets.

Token Reference

Last, WS-Security provides a mechanism to access external key bearing elements which could be located either in some other part of the document of completely outside the message. The <wsse:SecurityTokenReference> element is defined for this purpose.

A typical implementation of this element is as a child of the <KeyInfo> element when XML encryption and XML signature are used.

The `<wsse:SecurityTokenReference>` provides a generic and open mechanism to locate such element. Indeed these types of element don't provide a common referencing mechanism, and some of them even have closed schemas.

WS-Security Examples

Example

The example below is extracted from the WS-Security standard documentation and illustrates the use of a custom security token and its associated signature.

```
01 <?xml version="1.0" encoding="utf-8"?>
02   <S11:Envelope xmlns:S11="..." xmlns:wsse="..." xmlns:wsu="..." xmlns:ds="...">
03     <S11:Header>
04       <wsse:Security xmlns:wsse="...">
05         <wsse:BinarySecurityToken ValueType="http://fabrikam123#CustomToken"
EncodingType="...#Base64Binary" wsu:Id="MyID">
06           FHUIORv...
07         </wsse:BinarySecurityToken>
08         <ds:Signature>
09           <ds:SignedInfo>
10             <ds:CanonicalizationMethod Algorithm= "http://www.w3.org/2001/10/xml-exc-
c14n#"/>
11             <ds:SignatureMethod Algorithm= "http://www.w3.org/2000/09/xmldsig#hmac-
sha1"/>
12             <ds:Reference URI="#MsgBody">
13               <ds:DigestMethod Algorithm= "http://www.w3.org/2000/09/xmldsig#sha1"/>
```

```
14              <ds:DigestValue>LyLsF0Pi4wPU...</ds:DigestValue>
15            </ds:Reference>
16          </ds:SignedInfo>
17          <ds:SignatureValue>DJbchm5gK...</ds:SignatureValue>
18          <ds:KeyInfo>
19            <wsse:SecurityTokenReference>
20              <wsse:Reference URI="#MyID"/>
21            </wsse:SecurityTokenReference>
22          </ds:KeyInfo>
23        </ds:Signature>
24      </wsse:Security>
25    </S11:Header>
26    <S11:Body wsu:Id="MsgBody">
27      <tru:StockSymbol xmlns:tru="http://fabrikam123.com/payloads"> QQQ
</tru:StockSymbol>
28    </S11:Body>
29 </S11:Envelope>
```

Line 02 starts the SOAP envelop.

Line 03 declares the SOAP header.

Line 04 opens the WS-Security block.

Lines 05 to 07 declare a custom binary token.

Lines 08 to 23 specifies the signature for the content of the SOAP body located on lines 26 to 28.

Lines 18 to 22 provides key information through a <wsse:SecurityTokenReference> (line 20) which points to the token declared at lines 05 to 07.

Example

In this example WS-Security is used to sign a timestamp and the body of the message, and to encrypt an element of the body. Two X509 certificates are used. One is used for asymmetric encryption of the encrypted key, another for the signature.

```
01 <?xml version="1.0" encoding="utf-8"?>
02  <soap:Envelope>
03    <soap:Header>
04      <wsse:Security>
05       <wsu:Timestamp wsu:Id="T0">
06         <wsu:Created>
07           2001-09-13T08:42:00Z
08         </wsu:Created>
09       </wsu:Timestamp>
10       <wsse:BinarySecurityToken ValueType="...#X509v3" wsu:Id="X509Token"
EncodingType="...#Base64Binary">
11         ABCDEF....
12       </wsse:BinarySecurityToken>
13       <xenc:EncryptedKey>
14         <xenc:EncryptionMethod Algorithm="...#rsa-1_5"/>
15         <ds:KeyInfo>
16           <wsse:KeyIdentifier EncodingType="...#Base64Binary" ValueType="...#X509v3">
17             ABCDEF....
18           </wsse:KeyIdentifier>
19         </ds:KeyInfo>
20         <xenc:CipherData>
21           <xenc:CipherValue>...</xenc:CipherValue>
22         </xenc:CipherData>
23         <xenc:ReferenceList>
24           <xenc:DataReference URI="#enc1">
25         </xenc:ReferenceList>
```

```
26       </xenc:EncryptedKey>
27       <ds:Signature>
28         <ds:SignedInfo>
29           <ds:CanonicalizationMethod algorithm="http://...-exc-c14n#"/>
30           <ds:SignatureMethod algorithm="http://...#rsa-sha1"/>
31           <ds:Reference URI="#T0">
32             <ds:Transforms>
33               <ds:Transform Algorithm="http://...exc-c14n#"/>
34             </ds:Transforms>
35             <ds:DigestMethod Algorithm="http://...#sha1"/>
36             <ds:DigestValue>...</ds:DigestValue>
37           </ds:Reference>
38           <ds:Reference URI="#body">
39             <ds:Transforms>
40               <ds:Transform Algorithm="http://...exc-c14n#"/>
41             </ds:Transforms>
42             <ds:DigestMethod Algorithm="http://...#sha1"/>
43             <ds:DigestValue>...</ds:DigestValue>
44           </ds:Reference>
45         </ds:SignedInfo>
46         <ds:SignatureValue>.....</ds:SignatureValue>
47         <ds:KeyInfo>
48           <wsse:SecurityTokenReference>
49             <wsse:Reference URI="#X509Token"/>
50           </wsse:SecurityTokenReference>
51         </ds:KeyInfo>
52       </ds:Signature>
53    </wsse:Security>
54   </soap:Header>
55   <soap:Body wsu:Id="body">
56     <xenc:EncryptedData Type="http://www.w3.org/2001/04/xmlenc#Element" wsu:Id="enc1">
57       <xenc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-
cbc"/>
58       <xenc:CipherData>
59         <xenc:CipherValue>...</xenc:CipherValue>
60       </xenc:CipherData>
61     </xenc:EncryptedData>
62   </soap:Body>
63 </soap:Envelope>
```

Lines 05 to 09 define a timestamp of the message, referenced as T0.

Lines 10 to 12 contain a binary token, in the form of an X509 certificate.

Lines 13 to 26 contains information about the key used for encryption of element referenced "enc1" in the body. This key is encrypted with the public key contained in a certificate provided in the <KeyInfo> block at lines 15 to 19.

Lines 27 to 52 contain the signatures. The <KeyInfo> block at lines 47 to 51 refers to the X509 certificate contained into the binary token.

Lines 56 to 61 contains the data triple-des encrypted (line 57) with the key defined in the <EncryptedKey> block at lines 13 to 26.

Attacks on Web Services

Threats overview

Web Services aim at replacing most of the current communication and interaction protocols between the components of an application chain. They bring a large amount of new technologies and concepts which are not yet completely understood nor mastered.

Moreover the objective of providing an open, automated and distributed infrastructure naturally increases the potential attack surface and raises security concerns which were almost circumvented by current security technologies, would they be appropriately implemented.

As a consequence it is necessary to highlight the fact that Web Services today are at high risk. And as usual, understanding the threats and the attacks is the first mandatory step toward secure and reliable platforms.

Attacks on the basements

Attacks on XML

Introduction to XML Parsers

In order to process XML documents, it is necessary to use an engine which would be able to handle nested structure, potentially random order of elements and different representation of data. There is two main types of those "parsing" engine:

- SAX parsers, which are lightweight stream analyzing step-by-step engines;
- DOM parsers, which are more powerful as they are able to manage states and get the full understanding of the structure.

Depending on the parser used, vulnerability exposure will be different. However, none of them can be considered as fully secure.

XML Injection

Injections are a common way to have external pieces of code processed either by the server or other clients of the Web Service. These injections can be performed when user input is passed to an XML stream without proper validation nor sanitization.

Stream based parsers such as SAX parsers are particularly vulnerable to this type of attack as they feed variables as they process the data. Consequently, if an element appears twice in the same document they will use the last value they processed for this element.

Example

In the example below an XML injection is used to change the user role from User to Admin.

```
01 <UserRecord>
02     <ID>100374</ID>
03     <Role>User</Role>
04     <Name>John Doe</Name>
05     <Email>john.@doe.com</Email><Role>Admin</Role><Email>john@doe.com</Email>
06     <Address>1024 Mountain Street</Address>
07     <Zip>17000</Zip>
08 </UserRecord>
```

This kind of vulnerability can be found in pages which allow users to change their details such as email address.

The injection can be found on line 05 where the user injected some code which:

- Closes the first `<Email>` element;
- Adds an additional `<Role>` element with the modified value;
- Opens a new `<Email>` element in order to match the closing tag which will be added at the end of the user input.

Denial of service

There are several ways to generate denial of service attacks based on illegal XML documents.

Recursive payload

XML structure involves nested elements which may theoretically be found at any depth inside the document. Therefore parsing engines will usually follow the tree structure in order to process elements which can be found on the way. Usual documents will usually have nested elements down to a depth of four or five levels, up to a dozen in some specific cases.

However, it is trivial to create a malicious document which would involve hundreds or thousands of levels. In the case of parsers designed to fully understand and handle XML documents (such as DOM parsers) this could lead to an easy denial of service attack.

Oversized payload

Elements of XML documents can contain data of potentially any size, unless a limit has been specified in the schema and is enforced before the document gets processed. If these security measures are not properly applied it is possible to submit documents of several gigabits to the parser.

Such an attack is very efficient against parsers which entirely represent the document in memory before processing. Impact is excessive memory consumption which usually leads to a crash of the targeted resource.

Illegitimate values

XML parsers fill variables with potentially user provided input. Unsafe programming of variable content handling and processing, together with lack of user input validation, may lead to memory corruption, variable type mismatch or other types or errors which would result in interruption of the Web Service processes.

Application floods

Web Services are usually not designed to handle huge amount of requests. Indeed they are only used for processing operations while presentation and static content are provided by portals. Therefore the servers hosting such applications are not necessarily dimensioned to handle thousands of requests per second, such as usual web servers would be.

Therefore it is simple to overload the web service processes by simply sending a huge amount of legitimate requests and/or data.

CDATA Injection

The CDATA field has been defined in XML in order to make it possible to transmit non-legal characters. As a consequence data included into CDATA components will not be interpreted nor analyzed and will be processed as-is by the parsers.

As a consequence CDATA field can be used to:

- bypass schema validation;
- bypass or confuse intrusion detection engines;
- submit oversized data.

Example

In the example below a CDATA component is used to inject a persistent Cross-Site Scripting attack into a Blog entry. As CDATA component are not analyzed this attack makes it possible to avoid the detection of the patterns <SCRIPT>, </SCRIPT>, SCRIPT and SCRIPT and to prevent the stripping of the < and > signs.

```
<BLOG_ENTRY>
       <EMAIL>john@due.com</EMAIL>
       <TEXT>
              <![CDATA[<S]]>CRIP<![CDATA[T>]]>
              alert(document.cookie);
              <![CDATA[</S]]>CRIP<![CDATA[T>]]>
     </TEXT>
</BLOG_ENTRY>
```

Attacks on xPath

Introduction to xPath injections

xPath is the base component used for XML nodes research and manipulation. It is then used in a comparable manner to SQL in that it makes it possible to create queries and crawl into "XML databases". However, in terms of security xPath has two major drawbacks compared to SQL.

- xPath is a standard, as a consequence there is no need to port an xPath attack as all xPath implementations are identical (or should be);
- No access control is applied to the content of XML documents. Therefore xPath makes it possible to reference any part of an XML document, while SQL database usually restrict access to specific tables and columns.

The principle of xPath injections is similar to SQL injections. Indeed xPath queries can be modified in a similar way in order to collect additional (and theoretically confidential) information from the XML document or bypass authentication.

Simple xPath injections

Simple xPath injections are constructed with a single query similar to the `test' or 1=1 #` basic SQL injection, with some xPath-related specificites.

Example

An authentication mechanism would rely on the xPath:

```
//user[name='$login' and pass='$pass']/account/text()
```

This xPath expression retrieves the account relative to the user with $login/$pass credentials. If such combination doesn't exist, nothing is returned and the authentication fails.

A simple injection could be :

```
$login = whatever' or 1 or 'a'='b
$password = nervermind
```

As a consequence the xPath query would be:

```
//user[name='whatever' or 1 or 'a'='b' and pass='nevermind']/account/text()
```

The logical AND has higher precedence than the OR operator. Therefore the evaluation for credential is based on the expression:

```
(name='whatever' or 1) or ('a'='b' and pass='whatever')
```

The result is TRUE as the first predicates evaluate to TRUE, and the expression returns the list of all registered users.

Advanced xPath injections

The possibility to access any part of the XML document queried via xPath can be leveraged thanks to the '|' operator which can be considered as an equivalent to the UNION operator in SQL. Efficient exploitation of an unsecured user input can easily lead to the disclosure of a complete XML document.

Example

In the preceding example an xPath query was run to check the authentication credential against an XML user database.

In order to dump the full database another injection could have been used with the following user's input.

```
$login = whatever'] | /* | //user[ 'a'='b
$password = nervermind
```

The xPath query which will be evaluated is:

//user[name='**whatever'] | /* | //user['a'='b**' and pass='nevermind']/account/text()

First and third queries don't matter as the second one will retrieve the entire XML document. The only condition is that the queries are valid. As a consequence it is necessary to know the structure of the document, from a schema (XSD) or by any other mean.

Blind xPath Injections

Document discovery

This technique has been discovered by Amit Klein [BLINDXPTAH] and relies on the fact that xPath provides operators and functions which makes it possible to know the number of children nodes of a specific type are defined for an element.

The function used is the function count() used as described below.

`count(path/child::text())` returns the number of text elements

`count(path/child::comment())` returns the number of comment element

`count(path/child::*)` returns the number of elements

`count(path/child::processing-instruction())` returns the number of processing instructions elements

The identification of each element and its position can be performed thanks to the value returned by the count() function used with the expression below.

```
count(path/child::node()[position()=(x)] | path/child::text()[position()=(x)])
```

If the node x^{th} node of the element is a text node then the count function will return 1. Otherwise it will return 2.

As a consequence it is possible to identify nodes, their type and their names. By recursively performing this operation from the root of the document, the full structure can be discovered.

Booleanization

The injection itself cannot be performed as-is and needs an additional step before being sent to the server. This step is called Booleanization and aims at factoring the result of an xPath query into bits.

Indeed any string or number can be represented in Boolean value (B), and the value of the n^{th} bit can be extracted with a function like: `substring(B,n,1)`. Possible results are 0 or 1.

As the data of the child of a node at position N can be gathered with the expression: path/child::node()[position()=N], it is possible to get the name of the corresponding element with the expressions below:

```
substring(path/child::node()[position()=N],n,1)
```

with

```
0<=n<=length(path/child::node()[position()=N])
```

The result provides the value of the n^{th} bit of the data.

Blind injection

An injection can now be performed against any expression which would be exposed to a simple xPath injection vulnerability.

Example

In our first example the $login user input could be replaced by a Boolean expression E. The query constructed this way would have the structure given below.

```
//user[name='whatever' or E or 'a'='b' and pass='nevermind']/account/text()
```

If E is TRUE authentication is successful, otherwise E is evaluated to FALSE. Consequently we identify that the n^{th} bit of the data we are retrieving is 1 in the first case, 0 in the other.

Attacks against SOAP

Denial of service

Attacks on SOAP headers

SOAP messages are XML formatted. Consequently their processing is performed by parsers which are vulnerable to the same type of Denial of Service as those described earlier.

This exposure is leveraged by the fact that SOAP headers format and content are not strictly specified and the handling of specific values and operators highly depends on the implementation of the SOAP processor. For this reason SOAP headers are not strictly checked against XML schemas such as XSD and it becomes trivial to have recursive payload or oversized payload attacks launched and being efficient against SOAP engines.

Another kind of attack targeting SOAP headers simply consist in sending malformed SOAP requests. In some cases the fact that a specific field (such as the SOAP-Action field) is missing can generate Server errors and impact the processing of further SOAP messages.

DoS with SOAP attachments

External documents can be attached to SOAP messages. This capability is provided in order to allow the transport of data with unsupported type or size. This later criteria involves that SOAP attachments could be of any size, unless an explicit validation process enforces SOAP attachment size limits.

Otherwise SOAP engines can easily be flooded with messages containing oversized attachments. The impact would either be the exhaustion of system resources (such as memory if the message is loaded before processing), the slowdown (up to total interruption) of the traffic going through the SOAP-capable device, or a total interruption of service due to a system failure.

Simple flood attack on Web Services

SOAP is used as a transport layer for requests toward Web Services. It is then possible to send a high volume of legitimate requests targeting a specific SOAP interface. Would the processing of the request generate a sufficient load on the server, it would obviously leverage the impact of the attack by consuming excessive CPU and memory resources while requiring small firepower from the attacker.

Replay attacks

SOAP is a message communication protocol and is consequently stateless. It means that there is no native mechanism defined to establish and control sessions between clients and servers. Messages are handled and managed as a stream and no logical relationship can be established between them.

As SOAP does not verify the uniqueness of messages it processes, it is possible to send multiple time messages which have been intercepted during their transmission.

This mechanism can be applied in multiple scenarios such as authentication process, transaction between Web Service actors (client, intermediary providers and providers), and naturally allows spoofing attacks as source identification can be faked at the application layer.

Moreover authentication requests can be sent several times at any rate. Therefore SOAP provides a simple but efficient transport to perform brute force attacks on basic authentication mechanisms.

SOAP Fault exploitation

Whenever a SOAP request leads to an error the SOAP server generates a SOAP response containing a <soap:Fault> element. This element provides an error code (<faultcode> element) and an error description (<faultstring> element).

The later piece of information provided by the server can be responsible for the disclosure of sensitive information, such as the structure of an xPath query.

Example

The code below is that of a SOAP error generated by an xPath injection attempt. It discloses the xPath query processed by the server and provides valuable information regarding the XML document structure.

```
<?xml version="1.0" encoding="utf-8"?>

<soap:Envelope
             xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soap:Body>
    <soap:Fault>

      <faultcode>soap:Server</faultcode>

      <faultstring>Server was unable to process request. --&gt;
'/Users/User[attribute::Login=''' and attribute::Password='default']/*' has an invalid
token.</faultstring>

      <detail />
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

Moreover there is no standard defined to specify the content of the <faultstring> element. It is then possible to perform some fingerprinting of the different servers by submitting a voluntarily malformed SOAP request and analyzing the response provided by the server.

Example

In this example an empty SOAP request has been sent to a web service. The response sent by the server provides information regarding the technology used: apache/axis.

```
<?xml version="1.0" encoding="UTF-8"?>

  <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

```
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

    <soapenv:Body>
      <soapenv:Fault>
        <faultcode xmlns:ns1="http://xml.apache.org/axis/">
          ns1:Client.NoSOAPAction
        </faultcode>

        <faultstring>no SOAPAction header!</faultstring>
        <detail/>
      </soapenv:Fault>
    </soapenv:Body>
</soapenv:Envelope>
```

Attacking the Infrastructure

Service discovery

The purpose of discovery is to get information regarding the availability and the functions schema of a web service. Once this information is gathered it is possible to apply all the attacks described in preceding parts as all the necessary information are known:

- SOAP bindings;
- XML schema;
- Functions processes.

UDDI queries

UDDI servers behave like DNS servers as they provide the address of available services. However the information one can gather from these servers goes far beyond a simple address resolution. UDDI servers are designed to provide full information regarding the web services made available to anyone. Available data from UDDI servers are:

- The business list: a list and description of available business services
- The service list: a list of available web services
- The tModel list: a list of WSDL schema.

A simple set of queries is necessary to identify web service access point and WSDL schema, without any filtering and no authentication required.

Two steps are to be taken to reach this goal:

1. Get service key from any of the business, service or the tModel key from the tModel lists;
2. Request service details from the service or tModel lists.

Example

In this example the access point is discovered from the service list.

The first SOAP request contains:

```
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
<Body>
  <find_service eneric="2.0" xmlns="urn:uddi-rg:api_v2">
    <name>amazon</name>
  </find_service>
</Body>
</Envelope>
```

Extract from the response

```
<serviceInfo serviceKey="ba6d9d56-ea3f-4263-a95a-eeb17e5910db" businessKey="18b7fde2-
d15c-437c-8877-ebec8216d0f5">
<name xml:lang="en">Amazon.com Web Services</name>
</serviceInfo>
```

Service details related to the service identified by the service key retrieved are then requested.

The request is as detailed below.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
<Body>
  <get_serviceDetail generic="2.0" xmlns="urn:uddi-org:api_v2">
    <serviceKey>ba6d9d56-ea3f-4263-a95aeeb17e5910db</serviceKey>
  </get_serviceDetail>
</Body>
</Envelope>
```

An extract from the response we get clearly shows the access point for the web service

```
<businessService serviceKey="ba6d9d56-ea3f-4263-a95a-eeb17e5910db"
  businessKey="18b7fde2-d15c-437c-8877-ebec8216d0f5">
  <name xml:lang="en">Amazon.com Web Services</name>
  <bindingTemplates>
    <bindingTemplate bindingKey="1d3cf316-6b47-430b-9b8b-277a6e321e33"
         serviceKey="ba6d9d56-ea3f-4263-a95a-eeb17e5910db">
      <description xml:lang="en">
         The WSDL file that allows developers to make use of Amazon.com features on their
own site.
      </description>
      <accessPoint URLType="http">
          http://soap.amazon.com/schemas/AmazonWebServices.wsdl
      </accessPoint>
    </bindingTemplate>
  </bindingTemplates>
</businessService>
```

Other discovery methods

WSDL schemas are public documents made available to the world. Even though they are not referenced into a UDDI server, they can be found in several ways.

- They may have been referenced into a public Web Service search engine, such as seedka or xmethods;
- They may have been referenced into a generic search engine and can be found with search criterias like "site:target.com inurl:wsdl"
- They may be found through bruteforce discovery based on a dictionary of usual Web Service names and WSDL files.

Attacking WS-Security

Surprisingly WS-Security brought some insecure behaviours and vulnerabilities, which can be exploited in different ways.

XSLT Transform

Transforms are used by XML signature in `<Reference>` elements of the `<SignedInfo>` block as well as in the `<RetrievalMethod>` element of the `<KeyInfo>` block. They are defined for several purpose, such as nodes filtering (xPath filtering transform), canonicalization etc.

An optional transform is mentioned in the XML Signature standard: XSLT Transform. The capability of XSLT to modify the content of an XML document makes it a powerful tool for external content formatting. However most of the implementations of XSLT Transform provide system level functions. It means that it is possible to have the XSLT processor launching system commands.

This attack has been investigated and documented by Bradley W. Hill from Information Security Partners [XMLDSIGINJECTION].

As a consequence XML Signature raised a critical issue regarding the security of Web Applications, as it makes it possible to exploit the signature itself to have remote commands executed on the server.

Example

The example below details a <Transform> block which executes c:\Windows\system32\cmd.exe through Xalan, a widely used XSLT processor for the Java environment.

```
01 <Transforms>
02   <Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
03   <Transform Algorithm="http://www.w3.org/TR/1999/REC-xslt-19991116">
04     <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
05         xmlns:rt=http://xml.apache.org/xalan/java/java.lang.Runtime
06         xmlns:ob="http://xml.apache.org/xalan/java/java.lang.Object"
07         exclude-result-prefixes= "rt,ob">
08       <xsl:template match="/">
09         <xsl:variable name="runtimeObject" select="rt:getRuntime()"/>
10         <xsl:variable name="command"
11             select="rt:exec($runtimeObject,&apos;c:\Windows\system32\cmd.exe&apos;)"/>
12         <xsl:variable name="commandAsString" select="ob:toString($command)"/>
13         <xsl:value-of select="$commandAsString"/>
14       </xsl:template>
15     </xsl:stylesheet>
16   </Transform>
17 </Transforms>
```

The XSLT Transform is defined from line 03 to 16 and executes the system command defined by the exec function on line 11.

The scope of this vulnerability goes even beyond XML Signature as XML Encryption also uses the `<KeyInfo>` block (references as `<ds:KeyInfo>`) and the `<RetrievalMethod>` (referenced as `<ds:RetrievalMethod>`) which goes along with it.

Encryption Keys loop

The `<EncryptedKeyType>` is an extension of the `<EncryptedType>`. As such it embeds a `<ds:KeyInfo>` element, which can reference the encryption key via a `<ds:RetrievalMethod>` element.

It is then possible to create a loop of encrypted keys, referencing each other. This loop will lead to a Denial of Service against the web service performing the decryption of the data.

Example

In this example the encrypted key Key1 is encrypted using the encrypted key Key2, which in turns is encrypted using Key1… and so on.

```
01 <EncryptedKey Id='Key1' xmlns='http://www.w3.org/2001/04/xmlenc#'>
02   <EncryptionMethod Algorithm='http://www.w3.org/2001/04/xmlenc#aes128-cbc'/>
03   <ds:KeyInfo xmlns:ds='http://www.w3.org/2000/09/xmldsig#'>
04     <ds:RetrievalMethod URI='#Key2'
            Type="http://www.w3.org/2001/04/xmlenc#EncryptedKey"/>
05     <ds:KeyName>No Way Out</ds:KeyName>
06   </ds:KeyInfo>
07   <CipherData><CipherValue>DEADBEEF</CipherValue></CipherData>
08   <ReferenceList>
```

```
09     <DataReference URI='#Key2'/>
10   </ReferenceList>
10   <CarriedKeyName>I Said No Way</CarriedKeyName>
12 </EncryptedKey>

13 <EncryptedKey Id='Key2' xmlns='http://www.w3.org/2001/04/xmlenc#'>
14   <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc"/>
15   <ds:KeyInfo xmlns:ds='http://www.w3.org/2000/09/xmldsig#'>
16     <ds:RetrievalMethod URI='#Key1'
            Type="http://www.w3.org/2001/04/xmlenc#EncryptedKey"/>
17     <ds:KeyName>I Said No Way</ds:KeyName>
18   </ds:KeyInfo>
19   <CipherData><CipherValue>xyzabc</CipherValue></CipherData>
20   <ReferenceList>
21     <DataReference URI='#Key1'/>
22   </ReferenceList>
23   <CarriedKeyName>No Way Out</CarriedKeyName>
24 </EncryptedKey>
```

Although the standard warns against this kind of attack, it doesn't provide any security mechanism and recommends to implement a control mechanism which would monitor for abnormal resources utilization.

Reference traps

Another vulnerability identified but not solved by the standard is the malicious use of `<Reference>` elements. Indeed an external reference can be defined, either pointing to a very large resource or an infinite loop of redirect operations.

Both mechanisms are very simple to implement as they only require to have the `<Reference>` or `<CypherReference>` URI elements identifying either a large page or a page which will generate a 302 Redirect error code, redirecting to a page which generates a 302 Redirect error code to the first one.

Limitations of web application security mechanisms

Negative security model

Blacklist common limitations

The negative security model is based on signatures which are designed to identify malicious payloads which may be embedded into HTTP traffic. Their efficiency was proved against "out-of-the-box" exploits, mainly used in automated tools such as nikto [NIKTO] or nessus [NESSUS], and worms. In such cases they proved to be accurate and low resource consumers.

However, blacklists have shown many limitations which make them inefficient against several types of attacks.

Custom payloads

Many attacks are launched using common patterns which can easily be signed. However they can be easily transformed in such ways that a static signature will not be able to detect an attack.

As an example the the famous

```
[1] anything" OR 1=1 #
```

SQL injection, used to bypass filters or authentication can be changed into

```
[2] anything" OR 'whatever'='whatever' #
```

or

```
[3] anything"  or 'gotcha' in ('gotcha') #
```

These expressions will have the same effect, and it is obvious that it is not possible to create a comprehensive list of possible matches.

Additionally there are several evasion techniques which would be efficient in this case, such as whitespaces stripping or C-style comment injections.

Generic filters false-positives and negatives

Attempts to create generic blacklist filters often lead to increased and unacceptable proportion of false-positive alerts. Indeed such signatures should react on atomic components of an attack which, alone, may be part of a legitimate request.

Back to our previous examples, a possible try would be a signature which would block the pattern defined by the regular expression: `/or[^=]+=/i`

This signature would efficiently block the first two injections. However it would not block the third one. Moreover a URL such as /view.asp?orderID=12345, would also match and would be blocked.

Legitimate malicious payloads

In some cases applications use functions which natively enable the launch of commands. The main issue is that there is no internal filter regarding the commands which should be authorized by the system. As a consequence a malicious user can submit any command.

As an example the Oracle Secure Backup application was executing a command passed through the `rbtool` parameter upon logout of a user.

Then any URL with the format shown below could have been used to execute any command on the Oracle backup server.

```
http://<server>/login.php?clear=no&ora_osb_lcookie=aa&ora_osb_bgcookie=bb&button=Logout&rbtool=<command>
```

In these cases a filter based on negative security model should be able to sign all possible commands which are not to be considered as legitimate. This is obviously impossible.

"Zero-day" attacks

The drawback of blacklisting accuracy is their lack of efficiency against unpublished attacks, aka zero-day. As long as an attack is not known, it is not possible to write a filter. In some cases "standard" patterns can be helpful but, as seen above, these can be easily bypassed or generate false-positives.

Blacklist and Web Services attacks

Usual web application attacks remain valid for most web services as they only provide alternate communication path and transport format. In this environment SQL injections, cross-site scripting (mostly persistent ones), parameters tampering etc. will still hit the target. Indeed a `whatever" OR 1=1 #` inserted into an SQL command or a `<script>alert(document.cookie)</script>` stored in a database and displayed into a web page will have the same effects "as usual".

Therefore limitations previously detailed are still true. What is more, web services brought additional complexity and new attacks which cannot be signed and make the use of blacklist very limited.

Signature complexity

An additional issue brought by web services is the size and complexity of requests. As components of HTTP requests are contained into a URL and a limited and identified part of the message (header, POST parameters and cookies), their identification is quite straightforward and does not require the parsing and understanding of large pieces of data.

In the case of Web Services, request to the service are passed through SOAP messages, which in turn embed XML formatted requests. As a consequence the search for patterns requires extensive researches into potentially large data. This lead to a noticeable increase of resource consumption which may lead to unacceptable latency or even to denial of service attacks against the detection engine.

As an example, when a simple SQL injection could be simply detected in the URL:

```
http://hackme/login.php?login=1'%20OR%201=1%20--%20&password=1'%20OR%201=1%20--%20
```

It would be longer to find it in the SOAP request below:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:tem="http://tempuri.org/">
   <soapenv:Header/>
   <soapenv:Body>
      <tem:Login>
         <tem:loginID>1' OR 1=1 -- </tem:loginID>
         <tem:password>1' OR 1=1 --  </tem:password>
      </tem:Login>
   </soapenv:Body>
</soapenv:Envelope>
```

Moreover the quantity of possible format and transforms of transported data brings additional complexity for normalization and dramatically leverages the efficiency of evasion techniques.

New Web Services attacks

Only a very small amount of new vulnerabilities brought by Web Services can be efficiently signed with generic filters. It is the case of xPath injections, though they meet the same limitations as for SQL injections. However all the other attacks rely on the specific structure of the service and there is no way to create generic rules to define malicious or dangerous payloads in such cases.

For signatures used to address a specific vulnerability, they can be easily written and will prove to be quite efficient. The issue remains that web services are highly customized applications and it is quite unlikely that two different infrastructure will use the same services. Therefore specific filters are to be written by each organization once vulnerabilities have been identified in their application and before these vulnerabilities are patched. This significantly lowers the need and added-value for prevention devices.

Positive security model

Definition and implementation

The positive security model is designed to explicitly authorize data and actions. Any request which is not mentioned in the "whitelist" or which contains data that is not in the format specified in the list will be dropped.

This model can be very efficient for web applications as long as the structure of requests is well defined. However it is very rare that documentation for such applications is made available to security staff and current implementations usually goes along with a learning mechanism. This mechanism dynamically creates whitelists from observation of the traffic to the protected application. The duration of the learning phase depends on the complexity of the application and the relevance of the data used for learning.

When whitelists are successfully created, security policies are efficiently applied to mitigate injections-based threats and some parameters manipulations.

Positive security model and web services

The first difference between the implementation of a positive security model for web applications and web services is the location of the parameters.

In the first case parameters to be validated are located either in the URL (GET) or in the posted data. Therefore the scope of inspection, either for learning or filtering, is quite limited and no guideline is necessary.

Dealing with Web Service is a completely different issue as relevant data are contained nested elements of a document. As a consequence agnostic research of potentially exploitable components is a very complex and resource consuming task. Moreover formatting directives and different possible encoding make the characterization of allowed values more difficult, if not totally impossible.

Even in the case of normalized parts of the data, such as the WS-Security header, the flexibility of the standards and the lack of normative directives in most fields make generic learning and analysis quite unreliable.

Last, but not least, one of the main characteristics of web services is the capability to evolve automatically. It means that a specific function can be updated and accept new parameters. Client services will be aware of this change thanks to the WSDL of the service and may be able to adapt their behaviour very quickly. On the other hand a white-listing mechanism which has completed its learning phase will not be aware of the change and will then block legitimate traffic as it does not match with the pattern previously buit by the filtering engine.

Web Services Specificities

Canonization

Another common feature provided by Web Application security devices is normalization. These operations are quite efficient in order to detect and block filter bypass attempts, mostly based on insertion, replacement or encoding.

These evasion techniques can be applied to request for Web Services with the same results to be expected against the attack detection engine. However XML normalization engines will meet several issues which are not expected in raw HTTP applications.

The first one is, again, the quantity of data which is to be normalized. Indeed Web Applications only involve a URI and a few well located parameters, while the Web Services deal with full documents. The consequence is mainly an increased amount of resources needed to search and replace encoded or inserted data.

The second issue is the necessity to evaluate the type of XML element to normalize and the necessity (or even the possibility) to normalize them.

As an example, some references used into an XML documents may use relative path description. In this case the ../ or ./ string sequences are legitimate and should not be converted or suppressed. Such an operation would obviously make the reference unusable and would break the processing of the document.

External references

External references are also to be considered as a threat to Web Services as they can contain malicious data or voluntarily malformed structures. They can be considered similar in some points to the threat that would represent a malicious link on a web site, but with one noticeable difference. In the later case the target will be the client which establishes a connexion to the malicious web server. In the context of Web Services the target will be any component of the web service infrastructure processing the element which refers to the external resource.

Therefore a standard HTTP-like engine, protecting from request initiated by "external" clients, will not be able to efficiently detect and block attacks brought by an external reference.

Client / server identification

Another specific issue is brought by the communication streams between the actors of a Web Service infrastructure. The main difference with a standard HTTP communication stream is the fact that connections can be initiated by the intermediary providers. Indeed in common HTTP structures the client is initiating all the connexions to the servers. If an external resource is needed to load the full content of a page, then the client sets up a new connexion to the server hosting the resource.

In the Web Service world external information and data will be requested by the intermediary Web Service on behalf of the client. As a consequence such an actor has to be protected both from the client (which may be another intermediary Web Service) and from its provider which may return malicious data.

Improving current security mechanisms

Extending Canonization

Canonization of XML content is mainly needed to prevent detection evasion. There are two main types of operation to be performed:

- Standard XML canonization: apply XML canonization recommendation from W3C [XMLC14N];
- Additional canonization: perform transformations specific to attack detection operations and constraints.

It is important to notice that canonization should be performed on the full XML document before inspection, but the original data should not be changed as it would compromise the integrity of signed parts.

Standard XML canonization

The first need for canonization, close to that we have for Web Applications, is mainly made necessary to prevent detection evasion based on encoding, insertion and so on. String formatting as recommended by canonicalization standards is appropriate for such a task. As comments may also contain malicious data, the canonicalization mechanism implemented should include these elements as well.

The second need is the standardization of data before they are inspected. Indeed XML data can be represented in several equivalent ways. In order to have a reliable and efficient pattern matching process it is necessary to normalize data prior to their inspection. As this operation is by no mean related to the context but by the content itself inclusive canonicalization is to be used for this purpose.

Additional canonization

CDATA tags should be stripped out for inspection in order to make it possible to get full strings inspected. This will prevent common detection evasion techniques. However, as these tags may be inserted for a valuable reason they should not be stripped out from the data that will be passed to the web service once inspection is performed.

Last, the URI included in the XML document, such as external references or namespace identification, must also be normalized as they may contain common HTTP attacks which may be obfuscated. HTTP canonization of these URI will make inspection possible and efficient.

Pattern matching enhancement

Pattern matching engines remain efficient against most string-based attacks. However, common engines must be improved in order to stick with XML and Web Services specificities.

Main changes that should be implemented in these engines are:

- Perform identical analysis on requests and responses;
- Inspect the full content of the document;
- Include XML specific signatures.

Requests and responses inspection

As we previoulsy stated, one of the most noticeable change in the inspection process is the fact that malicious content may be found in requests from client as well as in responses from providers. As a consequence the analysis on data should be performed independently from their role in the current transaction.

Full document inspection

Attacks may be included in any part of the request or response:

- In standard HTTP fields, and mainly URI and headers;
- In XML data, which should be entirely inspected;

- In SOAP attachments, in which it is easy to include text-based attacks, externalized from the generic XML part of the document.

The last two points raise an issue in terms of performance and security for the inspection device. Indeed a malicious document could be crafted to consume excessive resources during the inspection process, especially in the case of oversized XML file or SOAP attachment.

Therefore the document inspection should be performed "on the fly" and the engine should handle the data in the same way as a SAX parser does. This is acceptable as long as canonization and additional mechanisms (such as WSDL validation) have been previously applied to the data to be inspected.

XML specific signatures

Attacks against Web Services rely on common techniques as well as specific ones. As a consequence new sets of signatures must be implemented in order to mitigate these attacks.

The main type of specific attack to mitigate is xPath and xQuery injections. In order to build an efficient detection mechanism it is necessary to define different kind of patterns.

Of course some other XML specific threats can be signed, such as the use of the <!ENTITY … SYSTEM …> tag or external references pointing to local addresses in requests coming from outside the local network.

Static patterns

These signatures are used to detect common injections as well as the call of functions. Indeed there is usually no legitimate reason for a request to call `position()` or `count()` functions, while these are used by blind injections.

In terms of common injections it is useful to define patterns for strings such as `' OR '1'='1' OR ''='` or `| /* |`. Although this mechanism offers a very limited security level it should be efficient against basic probe mechanism, automated security testing tools and worms.

These signatures can gain additional efficiency when built with carefully crafted regular expressions.

Example

The regexp below should efficiently most xPath injections of the type `' OR '1'='1' OR ''='` by signing the part in red in quite a generic way.

```
/('|")?\s*or\s*('|")[^'"]*('|")\s*=\s*('|")[^'"]*('|")/i
```

Graylisting

Static patterns have commonly known limitations: they cannot cover the full spectrum of possible attacks and their formatting/presenting variants, and extensive use of regular expressions impacts performances and increases the risks for false-positives.

Graylisting technique consists in applying weights to specific short and accurate patterns or characters. If a threshold is reached the string is considered as an attack and the XML document is blocked.

Typical patterns which should be included in this list are quotes, pipe, parenthesis, star etc.

The main issue is the definition of weights as erroneous values will either lead to false-positives or false-negatives. On the other hand this type of approach proved to be the most efficient in terms of security and scalability.

Web Services specific security

Application firewalling

Navigation through the services offered by different components of the Web Service infrastructure should be monitored and controlled to some point.

Protocol and language filtering

Some components are supposed to deliver only some specific type of service, such as UDDI or function call through SOAP. However, as traditional network firewall only filter on destination port, it would be possible to perform HTTP requests to access to resources which should not be served in this context.

Example

A Web Service may publish some public functions through SOAP requests. On the other hand this same server may host an administration web site, which should not be accessed with any HTTP request from outside the local network.

It is the necessary to implement a mechanism that, depending on the context, would be able to filter-out:

- Non-XML requests;
- Non SOAP request;
- Non UDDI requests.

Forceful browsing

Similarly some services or functions are provided by the server but should not be accessed from outside the local network, or even only by specific IP addresses on the network.

In this case, it is necessary to provide a mechanism which would restrict the access to such services based on IP addresses. Such a mechanism will also make it possible to avoid direct access of illegitimate clients to some services, which should only serve requests issued by identified intermediary providers.

This filtering mechanism should be built on top of the protocol and language filtering as it can be considered as a "extension" of the identification of SOAP messages.

Data and model validation

XML validation

There are two main characteristics which should be analyzed for potential threats against the Web Service: the node depth and the document size. Indeed, these two elements can be used to generate a denial of service attack against the server.

Document size

Document size limitation is self-explanatory, the only difficulty being to find the appropriate limitation. Such limitation can be defined based on the ratio filesize / memory usage or arbitrary set according to common observation of web services traffic.

In the first case a ratio 1/10 is acceptable for complex XML documents. It means that a 1MB file will occupy 10MB of memory. In the second case it is rare that XML documents larger than 100kB or 200kB are transmitted through SOAP.

Node depth

Document complexity impacts the CPU usage of the server processing it. As a consequence it is necessary to analyze documents in order to evaluate whether the depth of some nodes is acceptable or not. Defining an

accurate value for maximum acceptable node depth is not necessary. Indeed, in order to generate a denial of service it is necessary to create thousands of nested nodes, which is unlikely to happen in real cases.

As a consequence it is efficient and with no risk of false positive to set an arbitrary limit of node depth around 100.

A document which matches this condition should be dropped without further inspection.

WSDL validation

WSDL validation can be considered as a white list mechanism applied to XML document. The purpose is to apply positive security mechanisms to SOAP requests and responses received by the server, based on the service description provided by WSDL files.

This will make it possible to define filters which explicitly allow specific data types and length as input of functions provided by the Web server.

However this technique as a major limitation in the fact that WSDL are not designed with security in mind but interoperability. Therefore some argument definitions maybe too lose and would allow malicious data to be inserted into requests or responses and look legitimate.

As a consequence, once filters are created it is necessary to tune the settings which have been automatically generated. This operation ensures that a more strict check is performed on data passed to the server but requires some knowledge of the internals of the application to protect.

External references validation

Another XML / Web Services specificities is the capability to reference some content of the message which is located on other resources. This is a common case for keys used for signature validation or encryption, or for when XML signature is used to sign an external content.

These references point on a content which has to be validated in the same way as any XML content which will be submitted to the server. Therefore all the security checks described in this chapter (canonization, XML validation, signatures etc.) should be performed on these external documents as well prior to their access by the target server.

This can be done either by the security device when inspecting the document containing the reference or when the server retrieves the referenced resource. The first one has the advantage to perform all the security checks before the server starts the processing of the request. The main drawback is that the security device has to behave like a Web Service client and retrieve external documents. As a consequence it will consume a lot of resources and add a noticeable latency which may not be acceptable. Moreover will be exposed to potential attacks as well.

Data transformation

Sensitive data masquerading

Requests and responses may contain data which should not be disclosed outside a specific parameter. Although XML encryption may be used to handle such needs its implementation is not resource free and makes it necessary to deploy a much more complex infrastructure.

In many cases the replacement of data with alternate and meaningless ones. This is valid as long as the info contained in the data is no longer to be used by further processes and providers or requesters.

Example

It may be necessary to strip out credit card numbers contained in a request once this data has been processes. Therefore a simple replacement of the value of the element <CreditCardNumber> can be performed to provide the result below:

```
<CreditCardNumber>1230254678123667</CreditCardNumber>
```

Is transformed into

```
<CreditCardNumber>****************</CreditCardNumber>
```

SOAP fault neutralization

A more generic issue regarding sensitive information leak is the content of the <faultstring> element in SOAP responses. Indeed, as no specification or recommendation has been issued to define the content which should be displayed in this field, each server or Web Service use its own string. As a consequence becomes easy to perform fingerprinting of the target.

Moreover some <faultstring> elements even contain detailed information regarding the cause of the error. In some cases it may contain full xPath expressions or SQL request. This is very valuable information for an attacker as he will know exactly which is the structure of the targeted container (XML document or SQL database).

Therefore it is important to replace the content of these strings with a content which would not provide any valuable information to the client.

It is necessary to point out that replacing the content of <faultstring> elements will make it possible to identify the type of security device in use. However this should not represent excessive risk as long as the security device is secure.

Additional SOAP security

Message replay prevention

As SOAP is a stateless protocol, it is necessary to implement mechanism which would prevent message replay attacks. There are two possible impacts for this type of attack:

- Replay of intercepted request, usually authentication request, which makes it possible for a malicious user to get access to resources without the need for authentication;
- Denial of service attack, by flooding the Web Service with the same request sent repeatedly and at high rate.

In both case the detection should be based on the identification of similar messages received during a specific timeframe. The timeframe will vary according to the type of threat and implementation constraints.

Indeed DoS attacks are generated at relatively high rate and a short timeframe of a few seconds should be appropriate to identify abnormal repetition of the same messages.

However, in the case of intercepted requests replay scenarios the choice of the timeframe depends on the purpose of the request and the operations performed by the application which handles this request.

Example

In the case of a One Time Password authentication mechanism, which generates a new password every 10 seconds, an intercepted authentication request may be replayed during a timeframe of 10 seconds maximum. This request will be no longer valid once this delay is expired. Therefore 10 seconds would be an acceptable parameter for duplicate requests replay mitigation.

Obviously there is a limitation regarding applications which would not handle timing information and would accept a request issued days or months before.

SOAP attachment validation

The analysis of SOAP attachment must address several threats or security issues.

MIME Multipart analysis

The first one is the parsing of a multipart SOAP message. Indeed the usual SOAP messages format is does not include MIME parts. However such a message can be composed of a single part containing some malicious content.

Therefore a detection engine must be able to identify MIME parts boundaries, find the embedded XML document and analyze it properly.

Example

The SOAP message below is a Multipart document. The XML contains an xPath injection which must be detected.

```
MIME-Version: 1.0
Content-Type: Multipart/Related; boundary=MIME_boundary; type=text/xml

--MIME_boundary
Content-Type: text/xml; charset=UTF-8
Content-Transfer-Encoding: 8bit
Content-ID: <claim061400a.xml@claiming-it.com>

<?xml version='1.0' ?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:tem="http://tempuri.org/">
   <soapenv:Header/>
   <soapenv:Body>
      <tem:Login>
           <tem:loginID>whatever' or '1'='1' or ''='</tem:loginID>
           <tem:password>whatever </tem:password>
      </tem:Login>
   </soapenv:Body>
</soapenv:Envelope>
--MIME_boundary
```

Incapacity to properly handle SOAP messages which would be formatted this way would lead to easy detection evasion.

Attachment size limitation

Whenever a Web Service actor receives a multipart request it may attempt to process the attached data. In such case an oversized attachment would noticeably impact its resource utilization, especially regarding memory.

It is then necessary to implement a mechanism which would identify and block messages holding attachment larger than an acceptable size.

Conclusion

Web Services intend to be the next generation framework for the communications between heterogeneous systems. They provide an additional abstraction layer based on mature and relatively well mastered technologies. As a consequence they are found to be deployed in more and more infrastructures, but usually on a limited scope, yet. Indeed the current trend shows a noticeable increase of the projects size and complexity.

However the flexibility of the underlying standards has a major drawback: the necessary tradeoffs with normative directives. The lack of strictly defined and controlled environment makes it possible to implement almost anything almost anyhow. And this dramatically increases the possible attack surface against these services. Therefore securing such an application framework becomes a real technological challenge. Attacks are possible against the transport protocol, data can be stolen and even security functions can lead to system compromise.

Moreover, whereas the basics of the technologies involved are quite well known, the specificities used in each implementation are far from being understood. And this situation will get worst as current projects make use of larger set of functionalities over a larger scale. Naturally as technologies are no longer mastered at the required expertise level, security issues remain almost invisible. This lead to a fake impression of security, which is one of the worst possible scenarios.

Complexity and lack of awareness are the usual causes of major security breaches, and it becomes urgent to educate the persons who are involved in the design, the deployment and the security of Web Services. Because the understanding of technologies is a key element to properly evaluate the risks, to define an appropriate security policy and to implement efficient security solutions.

References

[BLINDXPTAH] Blind xPath injection – Amit Klein Sanctum/Watchfire - http://www.modsecurity.org/archive/amit/blind-xpath-injection.pdf

[NESSUS] http://www.nessus.org

[NIKTO] http://www.cirt.net/nikto2

[SAML20] Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0 – OASIS Standard - http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf

[SOAP12Adjuncts] SOAP Version 1.2 Part 2: Adjuncts (Second Edition) - W3C Recommendation – http://www.w3.org/TR/2007/REC-soap12-part2-20070427/

[SOAP12Framework] SOAP Version 1.2 Part 1: Messaging Framework (Second Edition) - W3C Recommendation – http://www.w3.org/TR/2007/REC-soap12-part1-20070427/

[SOAP12Primer] SOAP Version 1.2 Part 0: Primer (Second Edition) – W3C Recommendation – http://www.w3.org/TR/2007/REC-soap12-part0-20070427/

[WSDL20Adjuncts] Web Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts - W3C Recommendation – http://www.w3.org/TR/2007/REC-wsdl20-adjuncts-20070626/

[WSDL20Core] Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language - W3C Recommendation – http://www.w3.org/TR/2007/REC-wsdl20-20070626/

[WSDL20Primer] Web Services Description Language (WSDL) Version 2.0 Part 0: Primer – W3C Recommendation – http://www.w3.org/TR/2007/REC-wsdl20-primer-20070626/

[WSFL11] Web Services Federation Language (WS-Federation), version 1.1 - http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-fed/WS-Federation-V1-1B.pdf

[WSP15] Web Services Policy 1.5 – Framework - W3C Recommendation – http://www.w3.org/TR/2007/REC-ws-policy-20070904

[WSRM11] Web Services Reliable Messaging TC WS-Reliability 1.1 – OASIS Standard - http://docs.oasis-open.org/wsrm/ws-reliability/v1.1/wsrm-ws_reliability-1.1-spec-os.pdf

[WSSE11] Web Services Security: SOAP Message Security 1.1 – OASIS Standard Specification – http://docs.oasis-open.org/wss/v1.1/wss-v1.1-errata-os-SOAPMessageSecurity.pdf

[WSSP12] WS-SecurityPolicy 1.2 – OASIS Standard - http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/ws-securitypolicy-1.2-spec-os.html

[WST13] WS-Trust 1.3 – OASIS Standard - http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf

[XACML20] eXtensible Access Control Markup Language (XACML) Version 2.0 – OASIS Standard - http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf

[XML11] Extensible Markup Language (XML) 1.1 (Second Edition) – W3C Recommendation – http://www.w3.org/TR/2006/REC-xml11-20060816/

[XMLC14N] Canonical XML, version 1.0 – W3C Recommendation - http://www.w3.org/TR/2001/REC-xml-c14n-20010315

[XMLC14N-EXC] Exclusive XML Canonicalization – W3C Recommendation - http://www.w3.org/TR/2002/REC-xml-exc-c14n-20020718/

[XMLDSIG] XML Signature Syntax and Processing (Second Edition) – W3C Recommendation – http://www.w3.org/TR/2008/REC-xmldsig-core-20080610/

[XMLDSIGINJECTION] Command Injection in XML Signatures and Encryption – Bradley W. Hill, Information Security Partners - http://www.isecpartners.com/files/XMLDSIG_Command_Injection.pdf

[XMLENC] XML Encryption Syntax and Processing - W3C Recommendation – http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/

[XPATH20] XML Path Language (XPath) 2.0 – W3C Recommendation – http://www.w3.org/TR/2007/REC-xpath20-20070123/

[XQUERY10] XQuery 1.0: An XML Query Language – W3C Recommendation – http://www.w3.org/TR/2007/REC-xquery-20070123/