Analysis of security policy requirements in SOA-based technologies

Analiza wymagań polityki bezpieczeństwa w technologiach dedykowanych dla systemów SOA

Bartosz Brodecki, Jerzy Brzeziński, Sebastian Górecki, Piotr Sasak, Michał Szychowiak
# TABLE OF CONTENTS

Abstract ........................................................................................................................................................................... 7

1. Introduction ................................................................................................................................................................ 8

1.1 Service Oriented Architecture .......................................................................................................................... 8
1.2 SOA paradigm ................................................................................................................................................... 8
1.3 REST .................................................................................................................................................................. 8
1.4 WS architecture ............................................................................................................................................... 9
1.5 SOA implementations .................................................................................................................................. 9

1.2 Purpose and scope of this report ......................................................................................................................... 9

1.3 Organization of this report .................................................................................................................................. 10

2. Security in distributed systems .............................................................................................................................. 11

2.1 General security properties ................................................................................................................................ 11
2.2 Security in client-server model ......................................................................................................................... 12
2.3 Cryptographic mechanisms ............................................................................................................................ 13

2.2 Communication protection mechanisms .......................................................................................................... 15
2.3 Transport-level security .................................................................................................................................. 16
2.4 Message-level security .................................................................................................................................. 16

3. Analysis of security requirements for SOA systems ............................................................................................ 17

3.1 Security Policy ................................................................................................................................................ 19
3.2 Security policy model .................................................................................................................................... 19
3.3 Security policy information .......................................................................................................................... 21

4. Analysis of security standards for SOA systems ............................................................................................... 22

4.1 XML security extensions .................................................................................................................................. 23
4.2 XML Signature recommendation ...................................................................................................................... 23
4.3 XML Encryption recommendation .................................................................................................................... 25
4.4 XML Key Management Specification (XKMS) .................................................................................................. 26
4.5 Additional security considerations .................................................................................................................. 26

4.2 WS-Security / WS-SecureConversation .......................................................................................................... 27
4.3 WSS – SOAP Message Security .................................................................................................................... 27
4.4 WS-SecureConversation .................................................................................................................................. 31
4.5 WS-Trust .......................................................................................................................................................... 32
4.6 WS-Federation ................................................................................................................................................ 34
4.7 SOAP Messages with Attachments .................................................................................................................. 40
4.8 Additional remarks ........................................................................................................................................ 42

4.3 WS-I Basic Security Profile ................................................................................................................................ 43
4.4 Security Headers ............................................................................................................................................ 44
4.5 Attachments Security .................................................................................................................................. 46
4.6 Additional remarks ........................................................................................................................................ 46

4.4 WS-Policy / WS-PolicyAttachment .................................................................................................................. 46
4.5 WS-Policy model .......................................................................................................................................... 47
4.6 Policy Expression ......................................................................................................................................... 48
4.7 Policy intersection ........................................................................................................................................ 56
4.8 Additional remarks ........................................................................................................................................ 58

4.5 Web Services Security Policy Language (WS-SecurityPolicy) ........................................................................ 58
4.6 Protection Assertions ...................................................................................................................................... 58
4.7 Token assertions ............................................................................................................................................ 59
4.8 Security Binding Properties .......................................................................................................................... 60
### Table of Contents

1. **Security Binding Assertions** ................................................................. 61
2. **Additional remarks** ............................................................................ 64
3. **Web Services Enhancements (WSE)** .................................................. 64
4. **Turnkey security scenarios** ................................................................. 64
5. **Policy framework** .............................................................................. 66
6. **Security Assertion Markup Language (SAML)** ................................... 66
   - SAML protocols .................................................................................. 67
   - SAML bindings .................................................................................. 71
   - SAML profile ..................................................................................... 71
   - Example ............................................................................................. 71
7. **REST-related standards** ...................................................................... 73
   - Simple Authentication and Security Layer (SASL) ................................ 73
   - Simple and Protected Generic Security Service API Negotiation Mechanism (SPNEGO) .................................................................................................................................................. 73
8. **CORBA-related standards** ................................................................. 74
   - Security Domain Membership Management Service ................................ 74
9. **Policy definition languages** ............................................................... 76
   - Requirements ...................................................................................... 76
   - Extensible Access Control Markup Language (XACML) ..................... 78
   - Ponder ................................................................................................... 83
   - Ponder2 .................................................................................................. 88
10. **Security Policy Assertion Language (SecPAL)** ................................ 90
    - Related languages ............................................................................ 90
    - Additional remarks .......................................................................... 91
11. **Security specification languages for system modeling** .................... 92
    - UMLsec ............................................................................................. 92
    - SecureUML ...................................................................................... 94
12. **Concept and design of a new SOA policy language framework** ........ 97
    - SOA policy framework architecture .................................................. 97
    - General concept of a new SOA policy language .................................. 100
13. **Conclusions** ..................................................................................... 104
14. **Bibliography** ..................................................................................... 105
15. **Main acronyms and dictionary of terms** .......................................... 110
Abstract

This report presents an overview and analysis of various prerequisites for security policy environments compliant to the SOA distributed processing model. Several SOA-based technologies and security-related specifications, standards and recommendations are investigated. Also existing policy languages are evaluated. This analysis constitutes a foundation of forthcoming development of a new security policy definition language and framework suitable for the SOA and supporting all necessary requirements. Additionally, a preliminary concept of such a policy language is proposed.
1. Introduction

1.1 Service Oriented Architecture

**SOA paradigm**

Service Oriented Architecture (SOA) is a paradigm for developing services that may be distributed through a coarse-grained (loosely-coupled) environment encompassing different control (ownership) domains. Services are made accessible to consumers typically by providing descriptions for functions, related constraints and policies, and requirements for interactions (access or response). These descriptions need to be in a form in which their syntax and semantics are widely accessible and understandable. In order to process an interaction the consumer needs only to know how to access the requested service – i.e. the service interface, but not to the service implementation. An interaction is typically processed by series of message exchanges and invoked actions in a particular execution context – the set of technical and business elements that form a contract between service providers and consumers (service participants) [15].

For a service provider and consumer to interact with each other they have to be able to "see" each other. Such a visibility requires both participants to satisfy three following preconditions: awareness of existence of the other participant, communicational reachability and willingness to engage in the interaction.

A policy represents some constraints or conditions on the access to and use of a service or any other entity managed by any participant. Each of those constraints or conditions is expressed in the form of a policy assertion which may be evaluated and is ensured by policy enforcement mechanisms. Policies apply to many business-oriented or technical aspects of SOA, such as manageability, security, quality of service, hours of business, and so on. For the majority of SOA interactions it would be the most appropriate to include references to the policies associated with the service in the service description. While a policy represents the point of view of distinct participants, a contract is an agreement between the participants that represents their expectations about an interaction. Like policies, contracts can cover a wide range of aspects, like quality of service, service choreography, commercial agreements, etc. The execution context of a service interaction is the set of involved entities, technical aspects (infrastructure), policies and contracts that are identified as part of an instantiated service interaction [15].

Nowadays, the SOA paradigm is most broadly reflected by two still emerging technologies – REST and Web Services.

**REST**

From a certain perspective, REST (Representation State Transfer [9]) is a simple evolution of World Wide Web. It defines a more constrained architectural style to build Web applications, by imposing a few communication constraints: objects in the system are identified with Uniform Resource Identifiers (URIs), resources can be represented in a variety of widely-understood data formats (e.g. XML, HTML, CSS, etc.), and
representations are exchanged via protocols that use URIs to identify and directly or indirectly address the agents and resources (e.g. HTTP, FTP, SMTP, JMS, IIOP, etc.). Most important properties of well-formed URIs include conciseness, durability and predictability. As the REST architecture focuses on resources, it is sometimes referred as Resource Oriented Architecture (ROA [49]). So called RESTful services are required to provide uniform interface semantics (to create and retrieve the resource descriptions) rather than proprietary or application-specific interfaces, and manipulate resources only by the exchange of representations. This uniform interface consist solely of the operations defined by the HTTP protocol standard [10], which are: GET to retrieve a resource representation, PUT and POST to create or modify a resource, and DELETE to delete a resource. According to the REST concept, the GET operation must be idempotent – it should never change the state of a resource (comparably to a read-only method). Moreover, this architecture requires services to be stateless, i.e. every interaction must be isolated from any others. Among other features, the application statelessness allows to benefit from the current Web infrastructure including proxies and cache servers.

**WS architecture**

Web Services (WS) are another outcome of the WWW evolution, intended to provide a large number of standard means of interoperating between different software applications, running on a variety of platforms [5]. WS participants interact using SOAP [11] messages. SOAP is actually a general purpose data exchange protocol based on XML [6] typically carried over HTTP protocol. XML offers a standard, flexible and inherently extensible data format. It became therefore the key technology of WS, reducing the burden of deploying interoperable service descriptions, data representations and communication syntax. Along with XML itself, concepts of XML Schema and XML Namespaces play significant role in the WS world. From its part, SOAP provides a standard, extensible framework for packaging and exchanging XML messages. Web Services Description Language (WSDL [8]) provides an explicit way to describe service interfaces, including abstract description of messages exchanged between agents as well as bindings to concrete network protocols and message formats. WS architecture also requires a *service discovery* facility that may be implemented as a registry (such as UDDI [30]) in which the service provider agent publish the service description.

**SOA implementations**

There are at least few significant implementations of the SOA paradigm available on the market today, including components of such well-known platforms as IBM WebSphere [41], Oracle SOA suite [31], Microsoft Windows Communication Foundation (WCF [43]), among the most representative. Parallel report TR-ITSOA-OB7-5-PR-09-01 [73] describes these implementations, fleshing out the details concerning security issues.

**1.2 Purpose and scope of this report**

The main goal of this report is to specify and analyze the current state of development of security-related concepts and technologies adequate for SOA systems, with special attention on languages intended for describing security policies.

To accomplish that goal we need to focus first on more general concepts that make security requirements for SOA be distinct and particular, and therefore demand a research
for new solutions. Such a research has already been conducted for some time now and has led to a significant number of results, several of rather preliminary status, along with a few apparently more mature. This state-of-the-art must be carefully recognized and investigated. Different organizations, such as World Wide Web Consortium (W3C [20]), the Organization for the Advancement of Structured Information Standards (OASIS [16]) or Web Services Interoperability Organization (WS-I [19]), targeted to guide the SOA and WS infrastructure development, have already provided sets of standards and best-practices for various issues concerning SOA, such as communication protocols, service management or policy support, among others. The existing propositions should be analyzed in order to indicate further work.

1.3 Organization of this report

This report is organized as follows. Section 2 gives a brief insight on security of distributed systems. The most important concept and definitions are presented to sketch the background of this report. In section 3 we describe the most important issues of SOA that rise new security concerns and therefore require specific solutions to security problems in SOA-based environments. Then in section 4 we present an overview of efforts aimed at providing a standardized framework to solve those specific SOA security problems. It will cover such commonly accepted standards as WS-Policy, WS-Security, WS-Security Policy, WS-I Basic Security Profile or Security Assertion Markup Language, among others. Analysis of currently developed languages for security policy description is presented in section 5. This analysis leads to a new proposal for fully SOA-compliant security language, which concept and design principles are discussed at the end of section 5. Finally, section 7 gives some insights on further research, concluding this report.
2. Security in distributed systems

2.1 General security properties

In this section we will describe the most fundamental concepts of distributed security: i.e. authentication, authorization, confidentiality, integrity, availability, audit and non-repudiation. All these concepts constitute basic building blocks for final security properties that a security management must supply.

Security management mechanisms usually distinguish the following essential entities:

- **Resource** which is to be secured (this may be data stored in a database, a user account, a SOA service or communication facility, etc.).
- **Principal** – subject accessing the resource (a user or a process).
- **Policy** – meta-data document (perhaps in machine-processable language) describing rules for managing secure access to resources by principals (access permissions, etc.).
- **Policy guards** providing mechanisms by which the resource is secured according to a corresponding policy, roughly divided into two categories: Policy Decision Point (PDP) and Policy Enforcement Point (PEP).

By **authentication** we mean a verification of a principal’s identity. In a distributed environment, mutual authentication of both communication participants may be necessary.

**Authorization** mechanisms control the access to resources. Policy determines the access rights of a principal requesting the access.

**Confidentiality** is a property that restrict access to a resource only to authorized principals. For instance, it prohibits obtaining (reading) unauthorized information.

**Integrity** property guarantees that resources will not be altered by an authorized principals. For instance, it ensures that a message will not modified during transmission without detection.

**Authenticity** property ensures the origin of the secured resource (i.e. provides a confirmation about the identity of the author of the resource).

**Availability** property states that a resource will always be accessible to authorized principals.

**Audit** trails are required to trace resource access and principal behavior. They are key tool for ensuring integrity through verification. As it is not always possible to prevent the violation of required security properties, an audit may at least detect such violations and trigger necessary incident-response operations.

**Non-repudiation** is a property that prevents a resource originator from false denial of having created this resource, and a principal accessing a resource from denying to have
performed the access. For instance, this property disallows a message sender to contradict he or she (or it) has sent that message (with a given content).

**Security in client-server model**

The dominant model of communication in distributed service oriented systems is client-server. Several security issues concerns this scheme. For illustration, let us consider a simple scenario depicted in Figure 1, where Client C issues a request to server S, and gathers a response send back by S. In this scenario, server S might want to control whether or not to process requests from client C – making access control decisions. To do so, S must be provided with the exact identity of the requester, i.e. with request authentication. Also client C, in its turn, might want to decide whether it can safely interact with S – here the concept of trust will be involved (along with response authentication).

![Figure 1. Client-server model](image)

C or S might want to make sure that, while in transit, their requests and responses are not modified by unauthorized entities (integrity protection) — or eavesdropped upon (confidentiality protection). S might want C to pay for the services used, which requires monitoring interaction for accountability and security audit. Monitoring can also be useful to provide non-repudiation – allowing here to prove that the request (and response) did take place in fact.

**One-way communication**

Some RPC models support “fire-and-forget” invocations i.e. where no response is sent back (e.g. CORBA’s one-way methods). This is also the case of SOAP-based Web Services, where, if an invoked service does not return anything, no response message is sent to the client [12]. However, this simplified form of the client-server communication model still requires solutions to all security concerns sketched above.

**Nested invocations and delegation**

In a service-oriented environment, the client-server relation is always associated with a particular invocation. For instance, when invocations are nested (Figure 2), S₁ may act as a server when it is invoked by C, and as a client when it invokes S₂. The second invocation follows from a request propagation – a request travels through an invocation chain from C to S₁ and then (possibly modified or a new one) to S₂. In this case S₁ acts as an intermediate. Invocation chains introduce new aspects to the security of distributed systems and make the security picture much more complex. For example, should S₁ use its own identity invoking S₂, or should it explicitly represent C, using its identity (in which case the request sent by S₁ may appear as originated from C)? The latter is known as credentials delegation. Delegation rises subsequent problems, as for instance, should C trust S₁ to use
the confided credentials? However, delegation allows for dynamic determination of principal identities, on run-time, which in turn, may facilitate policy definitions.

Figure 2. Nested client-server model

Authorization decisions may be taken by S₂ according to its local access policy and applying the received credentials. Though, a delegation of authority can be used to distribute the authorization decisions – S₂ will rely on the delegated remote authorization instead of its local policy.

A side-effect benefit of using delegation to propagate the identity of the initial principal through an invocation chain is a possibility of system-wide accountability for the principal’s actions.

**Cryptographic mechanisms**

Cryptography is commonly used to ensure several security properties, namely: authentication, confidentiality, integrity, authenticity, among the most important.

Cryptography allows to encrypt (or sign) a resource with the use of an encryption key, in a way that permits only authorized principals – i.e. owning a corresponding decryption key – to decrypt the ciphered resource (or verify the signature).

There are two general types of cryptographic mechanisms, depending on the relation between encryption and decryption key pair: symmetric-key and asymmetric-key encryption.

**Symmetric-key encryption**

In the symmetric-key encryption scheme the same key is used to encrypt and decrypt the resource. Thus, in order to ensure the confidentiality of encrypted resources the symmetric-key algorithms require both participants of the interaction (the one that encrypts data and the other that decrypts it) to possess the same key. Both participants must keep the key in secrecy and protect it from any disclosure. In general, this imposes key secrecy and key distribution problems. Moreover, using the same key on both sides of the interaction makes it unfeasible to ensure the non-repudiation property, as there is impossible to distinguish an encrypted resource originated at one side from the same resource originated at the opposite side.

**Asymmetric (public)-key encryption, digital signatures and certification**

Asymmetric-key algorithms use a pair of related (but distinct) keys: one for encryption, the other for decryption (although their roles can be reversed). Such a distinction allows to
differentiate the operations of the opposite sides of the interaction, easily enabling authentication, non-repudiation and resource authenticity.

Typically, the key pair is bound to the resource consumer, e.g. a receiver of cryptographically protected messages. In order to ensure the confidentiality, the receiver’s decryption key must not be accessible to any other principal. Thus, this secret key is called *private*. Since the other key, used to encrypt a message, does not permit to decrypt that message (only the corresponding private key does), the encryption key need not to be kept in secrecy to assure the confidentiality. On the contrary, it can be freely available for any possible sender willing to communicate with the owner of the secret corresponding key. Consequently, the encryption key is named *public* (and the overall scheme is often called public-key encryption). Hence, the asymmetric-key approach trivially solves the key distribution problem known from the symmetric-key encryption.

Exchanging the roles of the public and private keys (i.e. encrypting a resource with the private one) allows every other principal to verify the authenticity of the resource. Simply decrypting the resource using the author’s public key proves the origin of the resource, since only the original author could have used the legitimate private key when encrypting the resource.

In contrast to efficient symmetric-key algorithms the public-key algorithms have much higher complexity, therefore – besides the possibility to provide more security properties – they are avoided in situations when only confidentiality protection is required. From the same reason, instead of encrypting the whole resource which may be quite large, only a small digest of the resource is typically created and encrypted to provide the authenticity. This is known as *digital signature*.

Although the asymmetric-key encryption does not suffer from the key distribution problem of the symmetric-key approach, there still remain some problems concerning the public key distribution. For instance, how to ensure the authenticity of the public key owner. Commonly, *certification* of keys is accepted as a solution to this issue. Certification involves a trusted Certification Authority (CA), issuing (and revoking) principal’s certificates of public keys (or other crucial information as timestamp certificates). Authenticity of certificates is also ensured with the signature mechanism. Thus, hierarchies of CAs exist, in order to authenticate the certificates.

**Hybrid encryption**

Hybrid encryption combines the efficiency of symmetric-key encryption with native facility of key exchange offered by public-key encryption. Each time a secure communication is to be established, a new session key needed for symmetric-key encryption is first exchanged securely using public-key encryption schema. Then all subsequent message exchanges use symmetric-key encryption.

**Public Key Infrastructure**

Public Key Infrastructure (PKI) consists of several standards to deal with public key certification, exchange and use. The main goal of PKI is to provide a common platform for any CA hierarchy. A world-wide PKI commonly accepted in Internet, PKIX developed by the Internet Engineering Task Force (IETF), is using X.509 as a standard certificate format.
A set of standard protocols has been developed for the PKI. They include such specifications as Certificate Management Protocol (CMP), Online Certificate Status Protocol (OCSP), Time-Stamp Protocol (TSP), Cryptographic Message Syntax (CMS) and Public Key Cryptographic Standards (PKCS), with the most prevalent: PKCS#1 (RSA Cryptography Standard), PKCS#5 (Password-Based Cryptography Standard), PKCS#7 (Cryptographic Message Syntax Standard), PKCS#8 (Private-Key Information Syntax Standard), PKCS#10 (Certification Request Syntax Standard), PKCS#11 (Cryptographic Token Interface Standard), and PKCS#12 (Personal Information Exchange Syntax Standard), among others.

2.2 Communication protection mechanisms

Services based on message-passing communication are exposed to several types of threats, including the following:

- Message confidentiality and message alteration in transport – the message can be read by an attacker or modified partially or completely, extra information may be inserted. Message structure can be altered. If the message contains attachments, also the attachments can be manipulated. An antagonist can send messages to a service that, while well-formed, lack appropriate security claims to warrant processing. An antagonist can also alter a message to the service which being well formed causes the service to process and respond to the client for an incorrect request.

- Spoofing can be engaged to abuse trust relationship between service participants.

- Man-in-the-middle attack may compromise an intermediary and intercept messages in transfer, transparently for the ultimate participants.

In such cases it is generally imperative to use cryptography mechanisms such as encryption and digital signatures. Those mechanisms are essentially useful for:

- Mutual authentication of participants
- Message data protection against unauthorised access during transmission.
- Verification of the message origin and integrity.
- Non-repudiation of message dispatch and reception.
- Confirmation of various security claims in the service interaction (security tokens carried by the exchanged messages).

Furthermore, other types of attacks, namely Denial of Service (DoS) and Distributed Denial of Service (DDoS) aim at preventing legitimate participants from establishing the communication. Those attacks take various forms and exploit weaknesses in the architecture, implementation or configurations of the system under attack. A possible scenario of such an attack may strive for overloading an operating systems hosting a given service or a communication device routing to it. Paradoxically, security components themselves are often targets of DoS/DDoS attacks, as they usually add a significant overhead that may willingly be exploited by attackers.
Transport-level security

One of possible methods of protecting point-to-point communication is transport-level security scheme (Figure 3). In this scheme, messages exchanged by the communicating parties (client C and server S, in the figure) are transmitted through a cryptographic tunnel (across a communication network “cloud”) established between them at the underlying transport-level\(^2\). Establishing such a tunnel usually requires authentication of both sides, which may be, at least potentially, combined with principals authentication, accomplishing this way identity verification of the service and its consumer. The tunnel may provide confidentiality protection with encryption of all transmitted data, or integrity protection using digital signatures, or both.

![Figure 3. Transport-level security](image)

The most commonly known implementation of transport-level security is SSL/TLS protocol suite. It provides several security properties: authentication of both parties of the communication, confidentiality and integrity with data encryption, and key exchange. SSL uses a combination of secret-key and public-key cryptography to secure the data traffic.

Besides of SSL/TLS, other widespread examples of transport-level security mechanisms are IPSec protocol and Virtual Private Network (VPN) technologies.

In general, transport-level security render any intermediaries impossible to operate on the protected data. Any processing is possible only outside of the tunnel.

Message-level security

When message-level security is applied to protect communication, each individual message is separately encrypted or/and signed (Figure 4). The protection mechanisms are incorporated into the message, possibly selectively, allowing some parts of the message (as headers, for instance) unencrypted. It enables to process protected messages, also by intermediaries (proxies, firewalls, etc.), as long as the protection remains intact. Moreover, the protection itself may be applied by such intermediaries on any desired part of the route between sender and receiver.

\(^2\) The term “transport” here may represent actually network, transport or session layer of the OSI model.
Nowe technologie informacyjne dla elektronicznej gospodarki i społeczeństwa informacyjnego oparte na paradygmacie SOA

TR-ITSOA-OB7-5-PR-08-01-2.2 „Analysis of security policy requirements in SOA-based technologies”

3. Analysis of security requirements for SOA systems

Although the SOA paradigm is sometimes perceived to be just a natural next step in evolution of object-oriented distributed middleware systems, in our opinion it has at least some properties making many aspects of SOA very specific. In this section we will try to emphasize those particularities, with special attention to security issues.

Let us start with some trivial observations on applying the SOA paradigm directly to a large scale environment. At least the following ones seem to have very important influence on building SOA-based systems at both conceptual and implementational levels:

- Applications are independently developed in different control domains, but they provide a functionality accessible to other domains as well.
- There is a substantial amount of platforms and programming languages the applications can use to provide their functionality. This great diversity of technologies may not restrict the inter-application cooperation.
- Security cannot rely on its domain boundaries and cannot limit the inter-domain (cross-enterprise) collaboration.

Therefore, in order to successfully meet the expectations of today’s worldwide business applications, the following specific prerequisites for any SOA-based system must be of special concern:

1. *High interoperability* – any of existing service development technologies must be supported. Technology differences in implementations of interaction entities must not limit the interoperability. Any specific technical constraints of a service implementation must be made invisible from the outside, hidden by the service interface.

2. Possibility to combine services offered by possibly different providers to create one larger *composite* service, with a single external view (interface). This composite service can be further combined with another services, until reaching desired level of functionality. That allows for the flexible development of complex business applications.

Examples of well-known message-level security mechanisms are Secure Multipurpose Internet Mail Exchange (S/MIME) or Pretty Good Privacy for electronic mail services.
3. Compliance to widely-adopted standards for implementing SOA infrastructure and services, i.e. to describe services, access services, manage interactions, make and enforce policy assertions, etc. Only the use of widely-adopted standards (preferably open standards) of interface definition, communication and management can ensure adequate level of compatibility between different entities involved in a SOA interaction. None of the formerly proposed distributed processing paradigms has been as clearly depending on such standardization as SOA is.

4. Loose coupling – integration of applications and orchestration of business processes across multiple processing platforms and control domains must be ensured. Management of the execution context that span over a large number of independent domains is a real challenge. Obviously the main key to fulfill this requirement is, again, use of (open) standards.

Consequently these prerequisites imply the following constrains in the context of application security:

- High interoperability and service composability requires the security mechanisms to conform to existing standards to ensure all security properties, including authentication, authorization and confidentiality of data processing and communication, among others. This covers for instance authentication token generation procedures, encryption algorithms and keys negotiation, certification infrastructure, access control information and decision support, etc.

- All involved security mechanisms must be accessible through widely-adopted communication facilities and protocols, like HTTP (with MIME) for RESTful services or XML-based protocols for WS-based systems.

- Loose coupling of the application interactions requires the security mechanisms to overcome the boundaries of single control domain. Inter-domain solutions are necessary to maintain security properties along with data traversal through the whole system.

- Security policy must be formatted in a some commonly accepted syntax, and some well-defined mechanism must be provided to retrieve the policy among the large-scale distributed system, in order to enable the agreement between both service participants: service provider and service consumer. Policy scope and enforcement becomes much more distributed, in the sense that it span over a spread heterogeneous environment, across various platforms.

- As most of service interactions are message-based, message-level security protection must be involved. Policy enforcement mechanisms would probably take form of intermediaries, thus enabling transparent fulfilment of policy requirements (for instance cryptographic protection of message content or removal of messages violating the policy). A message may travel thought multiple intermediaries until it reaches its destination. Message-level security in clearly indispensable in such cases.

- In case of complex communication patterns (as transactions) it is important to ensure that the business process was properly executed as a whole.

- Authentication and authorization mechanisms will operate on numerous cross-domain identities, using various identity representations and verification schemes adopted by each domain. A unified identity schema and representation will be strongly desired. Distributed trust relationship may come on the scene, allowing to use single federated
identities for principals originating from distinctive but entrusted domains. Distributed trust will also permit to make authorization decisions basing on delegation of principal credentials or delegation of authority.

- Secure service discovery must enforce policies governing publication and discovery of services. Although neglected today, his must be of great concern in SOA, where autonomous (machine-based) discovery is likely to extremely surpass human-assisted discovery, so machines must make a judgment about a service is trustworthy or not. Perhaps, in such cases, supplementary decisions are to be expected after autonomous selection of candidates, as for instance, checking a third-party authority for independent information about them. Thus, brand new language constructions for expressing secure discovery policy are required.

- Privacy and data-leak prevention may restrict interactions a service consumer can make to a limited set of services (service classes). Classification mechanisms are required and supported by a policy language.

In the remaining part of this section we focus on new challenges for security policy in SOA-based environments.

### 3.1 Security Policy

A service provider may define conditions under which the service is provided. For instance, this conditions may include such constraints as the requirement of successful authentication of the service consumer before processing his request or confidentiality protection of the content of a response message, etc. On the other hand, also a service consumer can impose some constrains on the service interaction. It may specify its own capabilities and requirements concerning the protection of the interaction. The set of all these conditions constitutes a security policy of a service. A policy is expressed in the form of policy assertions representing particular security conditions.

In SOA the policy (policy assertions) must have a representation intelligible to all the participants and must be accessible to them in a well-defined way.

**Security policy model**

In general, the most fundamental entities related to security are:

- a target that is to be protected (e.g. resource or service)
- a policy which defines what protection must be applied to these targets (governed at a Policy Administration Point)
- policy subject performing actions on targets (e.g. principal or client issuing access requests)
- mechanisms by means of which targets are protected (provided by Policy Decision Points and Policy Enforcement Points).

---

3 Actually, the notion of policy can be successfully extended beyond the scope of security only, into more general dependability policy (including such additional aspects of processing dependability as reliability – see WS-ReliableMessagingPolicy [52]). However, in this report we focus only on security-related aspects.
The Policy Decision Point (PDP) makes decisions about granting requested access to resources, issued by policy subjects. The Policy Enforcement Point (PEP) is responsible for generating decision requests to the PDP, executing decisions received from the PDP and piggyback security assertions required by PDP to service interaction messages. The Policy Administration Point (PAP) is used to define policy rules concerning access to resources. Rules defined by PAP are stored in policy repository – Policy Information Base (PIB).

This separation of basic policy elements isolates policy information from implementation, significantly simplifying policy system management.

Figure 5 gives a graphical representation of the considered policy model, with policy information data-flow.

![Policy data-flow model](image)

Figure 5. Policy data-flow model

Typically, the authorization decision can be either “allow access”, “deny access”, or “no policies apply to this access request”. If the PDP is unable to evaluate the policies or to retrieve required information for some reason, it may return an error instead of an authorization decision.

Policy enforcement mechanism would be implemented in a SOA environment most likely as intermediaries which intervene into the communication line in input and output directions. The pipelined processing of interaction messages fully conforms to the SOA architecture. The PEPs can then be treated as filters that perform transformations of the input/output messages during they flow to and form a service application. In input and output processing pipelines a number of policy assertions can be added or removed to/from interaction messages. Similarity, other operations can be performed in such a pipeline, as depicted in Figure 6.

---

4This model conforms to (and extends) the basic model proposed in ISO [67] and IETF [68] standards.
Security policy information

Since in SOA we consider independent applications interacting across different administration domains, it is important for a policy information to contain all information necessary to ensure required degree of security without scarifying application interoperability. In our opinion, the following types of policy rules must be supported:

- **Restrictions** – rules authorizing subjects with specific rights on targets. They define access control policy. Restrictions are provided by the PDP which grants or denied issued access requests according to restriction rules. Then, the PEP enforces PDP’s decision.

- **Obligations** – by which a target service provider or consumer may express what requirements a subject is expected to fulfill in order to accept its request. This may include for example, requirement to protect communication confidentiality with one of accepted cryptographic algorithms or to authenticate the request with an acknowledged signature standard. Obligations are opaque to the PDP. It is the responsibility of the PEP to understand and act on any obligations.

- **Capabilities** – by which a subject provides a list of alternative mechanisms it can provide to fulfill requirements. For example a subject informs that it is able to provide confidentiality protection with only a given subset of cryptographic algorithms or is using only a selected signature standards. Capabilities are defined by the PDP but it is the PEP who provides them to the opposite side of a service interaction.

In spite of access control, usually provided by current policy languages, obligations and capabilities are also truly necessary in SOA-based communication to allow both service participants to agree on one of possible alternative security requirements. Expressing obligations and capabilities of both sides of the service interaction is imperative for automatic context negotiation in veritable SOA environments, where interactions (very likely nested) are established (and contract parameters decided) dynamically, without any
human intervention or coordination, and where security requirements may probably be independent and separate from the service interface, and can change autonomically.

All kind of policy rules should be expressed in a maximum simplified syntax, to facilitate full interoperability.

We will refer to the above policy model and requirements in the following analysis of security standards (Section 4) and policy languages (Section 5), as well as in further concept of a new language and its framework (Section 6).

4. Analysis of security standards for SOA systems

Since the SOA security policy is required to operate in an extremely heterogeneous environment where a substantial quantity of security-related solutions influence service interactions, it is necessary to carefully investigate their characteristics.

There is a significant number of proposed standards, framework specifications, recommendations, guides and best-practices documentation concerning secure communication in SOA-based systems, most often Web services in practice. Their main goal is to provide a common basis for interoperability among multiple SOA products. Most of those standards started as proprietary industry initiatives, and finally has been transferred to standard bodies such as W3C, WS-I or OASIS. This chapter aims at giving a brief overview of those documents, as a basis of any further discussion about SOA security state-of-the-art.

Since Web services are inherently based on XML, we begin with description of two XML extensions: XML Signature and XML Encryption.

This report uses a set of XML namespaces in the examples. The list of used XML namespaces is presented below:

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Namespace</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsd1</td>
<td><a href="http://schemas.xmlsoap.org/wsdl/">http://schemas.xmlsoap.org/wsdl/</a></td>
<td>WSDL [8]</td>
</tr>
<tr>
<td>wss1e</td>
<td><a href="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd</a></td>
<td>WS-Security [14]</td>
</tr>
<tr>
<td>wsu</td>
<td><a href="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd</a></td>
<td></td>
</tr>
<tr>
<td>xs</td>
<td><a href="http://www.w3.org/2001/XMLSchema">http://www.w3.org/2001/XMLSchema</a></td>
<td>XML Schema</td>
</tr>
</tbody>
</table>

Table 1. The list of used XML namespaces.
4.1 XML security extensions

XML Signature recommendation

The XML Signature recommendation [25] specifies XML syntax and processing rules for creating and verifying digital signatures for any type of resources. That resource can be located within the XML structure that includes the signature or elsewhere. Any type of digital content (also XML) may be signed by this mechanism. Also, XML Signature can be applied to more than one resource. The signature itself can be placed also outside the XML document (detached signatures).

The XML Signature specifies signature element type (conformance requirements for element type is specified by XML Schema definitions). It also specifies a method of associating a key with referenced data.

An example of using the XML Signature is presented below:

```xml
<Signature Id="SampleSignature" xmlns="http://www.w3.org/2000/09/xmldsig#">
  <SignedInfo>
    <CanonicalizationMethod Algorithm="http://www.w3.org/2006/12/xml-c14n11"/>
    <SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1"/>
    <Reference URI="http://www.w3.org/TR/2000/REC-xhtml1-20000126/">
      <Transforms>
        <Transform Algorithm="http://www.w3.org/2006/12/xml-c14n11"/>
      </Transforms>
      <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <DigestValue> ... </DigestValue>
    </Reference>
  </SignedInfo>
  <SignatureValue>...</SignatureValue>
  <KeyInfo>
    <KeyValue>
      <DSAKeyValue>
        <P>...</P><Q>...</Q><G>...</G><Y>...</Y>
      </DSAKeyValue>
    </KeyValue>
  </KeyInfo>
</Signature>
```

Verification of digital signatures only works if they are performed on exactly the same bits as the signing calculations. If the surface representation of the signed data can change between signing and verification, then some way to standardize the changeable aspect must be used before signing and verification. The XML format is not absolutely strict, as allows for some diversification of the structure (i.e. two XML representations of the same document may “look” different, having still the same “meaning”). If an XML data is to be signed (and the signature verified), it has to be formerly transformed into a unified – canonical – form, in order to avoid any subsequent XML reformattting to influence the signature verification. Also for simple ASCII text, there are at least three widely used line ending sequences. Since the signed text may be modified from one line ending convention to another, then the line endings need to be canonicalized before signing and verification.

The SignedInfo element defines the canonicalization method. XML based canonicalization is provided with a XPath node-set. Text based canonicalization algorithms (such as CRLF and charset normalization) should be provided with the UTF-8 octets.
Signature algorithms are identified by URIs that appears as an attribute to the element that identifies the parameters of the signature processing (DigestMethod, Transform, SignatureMethod, or CanonicalizationMethod):

- **Message Digest algorithms**
  - Required SHA1 – http://www.w3.org/2000/09/xmldsig#sha1
  - Allowed other hash algorithms, that are better then SHA1, but when the WSS where published, they did not knew any better hash.

- **Message Authentication Codes**
  - Required HMAC-SHA1 –  
    http://www.w3.org/2000/09/xmldsig#hmac-sha1
  - DSA – http://www.w3.org/2000/09/xmldsig#dsa-shal
  - PKCS1 (RSA-SHA1) – http://www.w3.org/2000/09/xmldsig#rsa-shal

- **Signatures**
  - Required DSA with SHA1 (DSS) –  
    http://www.w3.org/2000/09/xmldsig#dsa-shal
  - Recommended RSA with SHA1 –  
    http://www.w3.org/2000/09/xmldsig#rsa-shal

- **Encoding**
  - Required Base64 – http://www.w3.org/2000/09/xmldsig#base64

- **Canonicalization**
  - Required Canonical XML 1.0 –  
    http://www.w3.org/TR/2001/REC-xml-c14n-20010315
  - Recommended Canonical XML 1.1 –  
    http://www.w3.org/2006/12/xml-c14n11

- **Transform Algorithms**
  - Optional XSLT Transform –  
    http://www.w3.org/TR/1999/REC-xslt-19991116
  - Recommended XPath Filtering –  
    http://www.w3.org/TR/1999/REC-xpath-19991116
  - Required Enveloped Signature –  
    http://www.w3.org/2000/09/xmldsig#enveloped-signature
  - Base64 – http://www.w3.org/2000/09/xmldsig#base64

Recommended key types are:

- RSA Key
- DSA Key
- PGP key
- X509 Certificate – it contains only name (identifier) of certificate
- SPKI – it may contains public key pairs or certificates
XML Encryption recommendation

XML Encryption recommendation [26] specifies a method for encrypting data and representing the result in XML. The data may be arbitrary like XML document, XML element, or XML element content. However, the result of encrypting the data is always an XML Encryption element which contains or references the cipher data.

Example of using the XML Encryption is presented below:

```
<EncryptedData Id='ED' xmlns='http://www.w3.org/2001/04/xmlenc#'>
  <EncryptionMethod Algorithm='http://www.w3.org/2001/04/xmlenc#aes128-cbc'/>
  <ds:KeyInfo xmlns:ds='http://www.w3.org/2000/09/xmldsig#'>
    <ds:RetrievalMethod URI='#EK' Type='http://www.w3.org/2001/04/xmlenc#EncryptedKey'/>
    <ds:KeyName>Sally Doe</ds:KeyName>
  </ds:KeyInfo>
  <CipherData><CipherValue>De$0Aq+D%B_EEF</CipherValue></CipherData>
</EncryptedData>

<EncryptedKey Id='EK' xmlns='http://www.w3.org/2001/04/xmlenc#'>
  <EncryptionMethod Algorithm='http://www.w3.org/2001/04/xmlenc#rsa-1_5'/>
  <ds:KeyInfo xmlns:ds='http://www.w3.org/2000/09/xmldsig#'>
    <ds:KeyName>John Smith</ds:KeyName>
  </ds:KeyInfo>
  <CipherData><CipherValue>xyRQ!@za*Ybc</CipherValue></CipherData>
  <ReferenceList>
    <DataReference URI='#ED'/>
  </ReferenceList>
  <CarriedKeyName>Sally Doe</CarriedKeyName>
</EncryptedKey>
```

Super-Encryption: Encrypting EncryptedData

An XML document may contain zero or more EncryptedData elements. EncryptedData cannot be the parent or child of another EncryptedData element. However, the actual data encrypted can be anything, including EncryptedData and EncryptedKey elements (i.e., super-encryption). During super-encryption of an EncryptedData or EncryptedKey element, one must encrypt the entire element [26].

Table of Algorithms

The list of categories of algorithms is presented below. Within each category, a brief name, the level of implementation requirement, and an identifying URI are given for each algorithm.

- **Block Encryption**
  - Required 3DES [http://www.w3.org/2001/04/xmlenc#tripledes-cbc](http://www.w3.org/2001/04/xmlenc#tripledes-cbc)
  - Required AES-128 [http://www.w3.org/2001/04/xmlenc#aes128-cbc](http://www.w3.org/2001/04/xmlenc#aes128-cbc)
  - Required AES-256 [http://www.w3.org/2001/04/xmlenc#aes256-cbc](http://www.w3.org/2001/04/xmlenc#aes256-cbc)
  - Optional AES-192 [http://www.w3.org/2001/04/xmlenc#aes192-cbc](http://www.w3.org/2001/04/xmlenc#aes192-cbc)

- **Stream Encryption**
  - There is no recommendation for stream encryption algorithms.
• Key Transport
  o Required RSA-v1.5 http://www.w3.org/2001/04/xmlenc#rsa-1_5
  o Required RSA-OAEP http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p

• Key Agreement
  o Optional Diffie-Hellman http://www.w3.org/2001/04/xmlenc#dh

• Symmetric Key Wrap
  o Required 3DES KeyWrap http://www.w3.org/2001/04/xmlenc#kw-tripledes
  o Required AES-128 KeyWrap http://www.w3.org/2001/04/xmlenc#kw-aes128
  o Required AES-256 KeyWrap http://www.w3.org/2001/04/xmlenc#kw-aes256
  o Optional AES-192 KeyWrap http://www.w3.org/2001/04/xmlenc#kw-aes192

• Message Digest
  o Required SHA1 http://www.w3.org/2000/09/xmldsig#sha1
  o Recommended SHA256 http://www.w3.org/2001/04/xmlenc#sha256
  o Optional SHA512 http://www.w3.org/2001/04/xmlenc#sha512
  o Optional RIPEMD-160 http://www.w3.org/2001/04/xmlenc#ripemd160

• Message Authentication
  o Recommended XML Digital Signature http://www.w3.org/2000/09/xmldsig#

• Canonicalization
  o Optional Canonical XML without Comments http://www.w3.org/TR/2001/REC-xml-c14n-20010315
  o Optional Canonical XML with Comments http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComments
  o Optional Exclusive XML Canonicalization without comments http://www.w3.org/2001/10/xml-exc-c14n#
  o Optional Exclusive XML Canonicalization with Comments http://www.w3.org/2001/10/xml-exc-c14n#WithComments

• Encoding
  o Required base64 http://www.w3.org/2000/09/xmldsig#base64

XML Key Management Specification (XKMS)

XML Key Management Specification complement XML Encryption and XML Signature to support distributing and registering public keys. XKMS allows the services to benefit from the PKI with the use of third-party CA components. Service consumers and providers contact the PKI with a SOAP protocol sending XML-formatted requests for issuing, retrieving, or revoking certificates.

Additional security considerations

A separate, yet important issues are cryptographic vulnerabilities of combining digital signatures and encryption over a common XML element. Some experts has suggested that
encrypting digitally signed data, while leaving the digital signature in the clear, may allow plaintext guessing attacks [26]. This vulnerability can be mitigated by using secure hashes and nonces in the text being processed.

How to actually use data encryption and signing is an application concern out of the scope of the XML Encryption and XML Signature specifications. However, several recommendations are proposed in this matter [28].

**XML processing**


Any information stored outside of the direct control of the user can be a source of insecurity e.g. included CSS style sheets, XSL transformations, entity declarations, and DTD definitions. For example, a "whiteout attack" to a master style sheet could make disappear selected parts of user documents, without modifying the user document or the style sheet it references. Thus, the security of any XML document is vitally dependent on all of the documents recursively referenced by that document.

Any change that produces a fatal error in a DTD could halt XML processing. Some sophisticated tricks, like changing attributes from being optional to required, can be difficult to track down. Perhaps the most dangerous option available is redefining default values for attributes: e.g., if developers have relied on defaulted attributes for security, a relatively small change might expose enormous quantities of information.

It is also possible to insert arbitrary and unauthorised text into documents by "entity spoofing". XML 1.0 permits multiple entity declarations, and the first declaration takes precedence. Thus, it is possible to insert malicious content where an entity is used, e.g. insert the full text of a large book in every occurrence of some typical entity (e.g. "--").

### 4.2 WS-Security / WS-SecureConversation

**WSS – SOAP Message Security**

The WS-Security – SOAP Message Security specification (WSS [14]) is designed to be used as a flexible basis for securing Web services within a wide variety of security mechanisms like PKI, Kerberos, SSL, and others. In particular, it specifies SOAP extensions for ensuring message confidentiality using XML Encryption and data integrity using XML Signature standards.

Actually, the WSS specification only provides building block that can be used in conjunction with other WS extensions and higher-level application-specific protocols to accommodate a wide variety of security models and security technologies. It describes three main mechanisms:

- ability to send security tokens as part of a message,
- message integrity protection,
• message confidentiality protection.

The following topics are outside the scope of the WSS:
• establishing a security context or authentication mechanisms,
• key derivation,
• advertisement and exchange of security policy,
• hot trust is established or determined,
• non-repudiation.

WSS introduces security tokens to assert the bindings between authentication secrets or keys and principal identities. For instance, a security token may be used by a service consumer as form of a proof-of-possession within an authentication scheme required by a service provider.

Figure 7. SOAP envelope with WSS Security Header

Message Protection

Protection of the message is provided by encrypting and/or signing a body, a header, or any combination of them. According to the WSS standard, XML Signature is providing message integrity, as it supports multiple signatures, potentially by multiple SOAP actors/roles, and is extensible well enough to support additional signature formats. On the other hand, XML Encryption is leveraging message confidentiality, as it supports additional encryption processes and operations by multiple SOAP actors/roles.

Messages should be rejected by recipient if they are containing invalid signatures, missing necessary claims or have unacceptable values.

The following example will be referred in subsequent description of WSS attributes:

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

Attribute `wsu:Id` represents a schema type of ID which is used to reference elements. When trying to locate an element referenced in a signature, the following attributes are considered:

- local ID attributes on XML Signature elements,
- local ID attributes on XML Encryption elements,
- global `wsu:Id` attributes on elements,
- profile specific defined identifiers,
- global `xml:id` attributes on elements.

The `wsu:Id` is used for identifying and referencing elements.

**Security header**

The `<wsse:Security>` header block involves security-related information targeted at a specific recipient in the form of a SOAP actor/roles. Each message may have multiple security header blocks targeted for separate recipients. There is no specific order of processing the sub-elements of Security headers.
Security tokens

The wsse:Security header is composed of security tokens. A simple example of a security token would be Subject Confirmation. It defines how signatures may be used and associated with security tokens as a form of claim confirmation.

All XML-compliant Security tokens use XML format, and all non-XML tokens should use wsse:BinarySecurityToken elements; in both cases two obligatory attributes are specified to define the actual token encoding:

- **ValueType** – indicates what the security token is.
- **EncodingType** – tells how the security token is encoded.

Token References

Token references may be used to locate key elsewhere in the message or completely outside the message. There are specific references defined in WSS:

- **Direct References** – allows including tokens using URI fragments and external tokens using full URIs.
- **Key Identifiers** – allows tokens to be referenced using an opaque value that represents the token.
- **Key Names** – allows tokens to be referenced using a string that matches an identity assertion within the security token. This is a subset match and may result in multiple security tokens that match the specified name.
- **Embedded References** – allows tokens to be embedded directly.

Signatures

The WSS specification allows for multiple signatures and signature formats to be attached to a message, each referencing different, even overlapping, and parts of the message. The WSS requires only that the signature validation must fail if:

- the syntax of the content of the element does not conform to this specification,
- the validation of the signature contained in the element fails according to the core validation of the XML Signature specification,
- the application applying its own validation policy rejects the message for some reason (e.g., the signature is created by an untrusted key).

The Signature Confirmation element may be included in the message signature of the associated response message.

Encryption

The WSS specification allows for encryption of any blocks by either a common symmetric key shared by both sides, or a symmetric key carried in the message in an encrypted form.
XML Encryption standard permits different `<xenc:EncryptedData>` elements referenced by the same `<xenc:ReferenceList>` to be encrypted with different keys.

Also header block encryption is supported. For that purpose `<wsse:EncryptedHeader>` has been introduced. This element can be super-encrypted and replaced by other encrypted header blocks (for wrapping/tunneling scenarios).

**Security Timestamps**

It is possible to define creation and expiration time for message. Time synchronization is not in the scope of the WSS. Only one `<wsu:Timestamp>` element in could be included in `<wsse:Security>` header.

**Error handling**

Some type of errors are specified in WSS:

- invalid or unsupported type of security token, signing, or encryption
- invalid or unauthenticated or unauthenticatable security token
- invalid signature
- decryption failure
- referenced security token is unavailable
- unsupported namespace

**Remarks**

Signing messages is indeed a big problem. Consider signing only some elements of a SOAP message, but no the whole message. It is possible to change order of elements in the message, add new part of the message, and possibility of use many signatures. However, it is possible to remove some part of the message with signature. Service consumer cannot check if it get the whole message.

We may sign the whole SOAP message. It makes it impossible to remove some part of the message. Adding new elements to the message should require a new signature of the whole XML message. Let us consider that each service participant add some new elements and each sign again the whole message. It is nontrivial to check the correct signature and witch signature to choose. Nevertheless, the WSS say it is recommends that applications signing any part of the SOAP body sign the entire body.

**WS-SecureConversation**

While message authentication is useful for a single request/response interaction, service participants that wish to exchange multiple messages should establish a `security context` in which the message exchange can be carried without continuous negotiation and actualization of security parameters. All the parameters of a security context shared among the communicating parties keep their values for the lifetime of a communications session. A good example of such a security context parameter could be the symmetric encryption
key used to protect the message. If the service interaction requires both participants to exchange a large number of messages, then instead of generating the cipher key for each message separately, the key value once generated may remain unchanged during the whole communication, saving a significant amount of processing power (per-message cost for key generation, key encryption, and key decryption), as well as network costs for transmitting the key every time.

Before being used, a security context needs to be created and shared by the participants. The WS-SecureConversation specification [29] defines three ways of establishing a security context:

- The security context is created by a WS-Trust Security Token Service (see WS-Trust on page 32 below).
- The security context is created by one of the communicating parties and propagated with a use of the WS-Trust.
- The security context is created through negotiation and exchanges also within the WS-Trust framework.

Typically a security context contains some shared secret used to derive actual keys for signing and encrypting messages. Using a common secret, the communicating parties may define a quite complex key derivations schemas. Although different algorithms can be used to derive keys, the WS-SecureConversation specification defines its own algorithm based on a subset of the mechanism defined for the SSL/TLS protocols.

A shared security context is identified by the SecurityContextToken element in the Security header entry in a SOAP message. The SecurityContextToken element and its child elements are defined in a namespace specified by the WS-SecureConversation. All messages exchanged during a given conversation have the same SecurityContextToken identifier. In contrast to the previously described WS-Security interaction, now the EncryptedKey element of a SOAP message is omitted (as the key needed to establish a secure conversation has already been negotiated when establishing a security context), therefore ReferenceList elements can be directly used to declare what message parts are to be decrypted by the recipient using the symmetric key known from the security context.

**WS-Trust**

The WS-Trust [48] language is used to extend the WS-Security mechanisms with additional primitives for issuing, exchanging and validating the security tokens (as credentials, in most typical case) within distinct trust domains. The main problem with validating credentials between two service participants, is to determine whether they can securely trust the asserted credentials received from the other party, especially when there is no common authority issuing the credentials (as Certification Authority) shared by them both. The WS-Trust defines methods of establishing and managing a trust relationship. WS-Trust enables also to support well-known trust models such as Kerberos or PKI.
Each service participant may establish trust in any way, but the WS-Trust distinguishes three types of trust:

- **Fixed trust roots** – each service participant has a fixed set of trust relationship.
- **Trust hierarchies** – build on the above trust, but it is allowed to build the hierarchies of trust so long as the trust chain leads to one of the known trust root.
- **Authentication service** – service consumer forwards tokens to the authentication service, which replies with an authoritative statement attesting to the authentication.

WS-Trust facilitates security token interoperability with the use of a **Security Token Service (STS)**. It is a web service that issues security tokens, i.e. it makes assertions based on evidence that it trusts, to whoever trusts it or to specific recipients. Consequently it allows a service participant to request a security token from a third trusted authority. As an example let us consider a company willing to do business with another company. A client application from a domain of the first company may authenticate itself to the STS that is trusted by that second company, and then request a SAML assertion. Further, when accessing a given service provided by the second company, the client application will submitted this assertion as security credentials.

WS-Trust defines a simple request-response protocol. When a service consumer sends a request (**RequestSecurityToken**, RST) to a STS it must first prove to be authorized to get the requested security token. This request can be submitted by a service consumer or by another party on the service consumer behalf. After successful authentication and authorization of the requester, the STC replies with a **RequestSecurityToken-Response** (**RSTR**) The request and response are typically exchanged using SOAP. The token may however be requested and obtained using any third-party or legacy protocol. This type of token acquisition is called **Out-of-Band Token Acquisition**.

**Negotiation and Challenge**

It is possible that WS-Trust mechanisms exchange messages for negotiation and challenges. The challenge mechanism was specified for authentication. Other types of mechanisms like negotiation, tunneling of legacy protocols etc. may be used in the same way. This mechanism is illustrated in Figure 8.
Token services may use renewal to rekey, responses may include a new proof-of-possession token, an entropy, a key exchange elements. If a previous security token with expiration is presented (and proven) for removal and the same token is returned with new expiration semantics.

Extensions to the `<wst:RequestSecurityToken>` elements are defined for transporting policies [33]. The policies may be used to specify desired settings for the requested token.

WS-Trust defines all XML-formed messages for exchanging trust relationship between service participants.

- Request of a Security Token:

```xml
<wst:RequestSecurityToken Context="...
xmlns:wst="...">
  <wst:TokenType>...
  <wst:RequestType>...
  <wst:SecondaryParameters>...
</wst:RequestSecurityToken>
```

- Response from STS with a Security Token:

```xml
<wst:RequestSecurityTokenResponse Context="...
xmlns:wst="...">
  <wst:TokenType>...
  <wst:RequestedSecurityToken>...
</wst:RequestSecurityTokenResponse>
```

The policies used by the STS are specified according to WS-Policy language and with the use of WS-SecurityPolicy standard (see section 4.5 on page 58 below). Policies are usually composed according to WS-MetadataExchange [50] and accessed with WS-Transfer protocol [1].

**WS-Federation**


The goal of this specification is to introduce the concept of federation in Web Services. The federation allows security principal identities and attributes to be shared across defined local domains (single organization boundaries), according to established federation policies. The federation polices define, among other things, trust and privacy requirements. In the web services environment in particular, the federation allows brokering service consumers’ identities and attributes to services without requirement of user interaction.

Federations can be created inside one organization or can span across organization administrative boundary. They can be dynamic (like ad-hoc environments), explicit
establishing a federation context for a principal requires either the principal’s identity to be universally accepted (i.e., its association is already established across trust realms within the federation context) or it must be brokered into a trust identity relevant to each trust realm within the federation context. The second case needs some identity mapping. The mapping carries out conversion of a digital identity from one realm to a digital identity valid in another realm. The mapping process is provided by an entity which trusts the first realm and has appropriate rights to speak to another realm or make assertions that the ending realm trusts. The identity mapping is typically implemented by an IP/STS (a service act that is either an Identity Provider or Security Token Service) when initially obtaining tokens for a service or exchanging tokens at a service’s IP/STS.

Figure 9. Examples of federations

Metadata model

Federation metadata describes settings and information about how the service can be used in the federation environment. Another type of metadata is communication policy which describes the requirements for web services messages sent to the service and WSDL description of the service. In the federation environment there is a need to discover the metadata and policies of the participants within the federation. The discovery starts with the target service, to which the requester wants to establish a new communication. The metadata endpoint reference (MEPR) is used by the requester to acquire all necessary information about the service (e.g., federation metadata, communication policies, WSDL, etc.). The process of obtaining the metadata is depicted in Figure 10.
Nowe technologie informacyjne dla elektronicznej gospodarki i społeczeństwa informacyjnego oparte na paradygmie SOA

TR-ITSOA-OB7-5-PR-08-01-2.2 „Analysis of security policy requirements in SOA-based technologies”

Figure 10. Acquiring service metadata

Trust Topologies and Security Token Issuance

WS-Security, WS-Trust, WS-Policy define together the basis for federation trust. The WS-Federation specification extends it introducing trust realm mechanisms across and within federations. Some typical scenarios of possible trust topologies are illustrated below:

Figure 11. An example of trust topology

In Figure 11, security tokens (1) from requestor’s trust realm are used to acquire security tokens from the service’s trust realm (2). These tokens are then presented to the service’s realm (3) to access the service.
Another example is presented below:

![Validation approach in trust topology](image)

In this example, the IP/STS entity acts as a validation service. The requestor acquires the token from requestor’s IP/STS (1,2) and then the token is provided to the service. The service uses its service’s security token service to understand and validate provided token (3).
In the final example illustrated in Figure 13 there is no direct trust relationship, but a third party exists, providing direct trust relationships for the requestor and service separately.

![Indirect Trust topology](image)

**Figure 13. Indirect Trust topology**

**Identity Providers**

Issuing and exchanging security tokens are proceeded by a Security Token Service (STS). Any web service can be a STS simply by supporting WS-Trust specification. There can be different kinds of STSs. One might only verify credentials for realm entrance while others can evaluate the trust of supplied security tokens. Providing digital identities can be example of typical STS functionality. This kind of security token service is named an *Identity Provider (IP)*. The IP performs authentication and can make identity claims in issued security tokens. The example of possible combination of an IP/STS is illustrated in Figure 14.
Attributes and Pseudonyms

There are situations when applications need additional information about the requestor and that information has not been already provided or there is no need to send that information every time along with the message in a security token. Such auxiliary *attributes* can store any additional information about the requestor required for access control, as an example.

In a single sign-on environments where multiple identities need to be automatically mapped, a *pseudonym service* can be useful. A pseudonym service allows a principal to have different aliases at the different services or in different realms. In situations when single identity can not be trusted as presented, the pseudonym service provides mapping between the identity and pseudonym for a particular principal. Pseudonym service can be implemented in IP/STS.

Additional remarks

Nevertheless federated identity is an important and very useful concept in SOA, some additional security issues requires special attention and care:

- All federation-related messages such as attribute, pseudonym and others should be integrity protected (e.g. protection at the transport layer or signing massages).
- The attribute service can store and/or process sensitive information such as financial or privacy related. Significant care should be taken to maintain high level of security of attribute service.
- Care should be taken during processing requests in federated environments. The processing should be well defined to mitigate potential information disclosure attacks.
- Significant care should be taken to maintain high level of security of the pseudonym service. The pseudonym service may contain sensitive information like security credentials used to proof identity. Such information should be encrypted during sending over communication channel and in any physical storage.
SOAP Messages with Attachments

Each SOAP message can be accompanied by several attachments. Such a message will have a MIME header and possibly multiple boundary parts. The primary SOAP envelope should be conveyed in the first MIME part. The SOAP Messages with Attachments specification (SwA [21]) describes how a WS consumer can secure SOAP attachments using WSS – SOAP Message Security for attachment integrity, confidentiality and origin authentication, and how a receiver may process such a message. The main goals of this specification are the following:

- Enabling to secure messages with all or selected attachments, using WSS.
- Allowing the choice of securing MIME header information exposed to the SOAP layer, if desired.
- Do not interfere with MIME transfer mechanisms, in particular, allow MIME transfer encodings to change to support MIME transfer, despite support for integrity protection.
- Do not interfere with the SOAP processing model – in particular allow SwA messages to transit through SOAP intermediaries.
- Specification does not provide information how to support signing and/or encryption of portions of attachments. XML Signature and XML Encryption may be used to accomplish this task.

SwA specifies two MIME mechanisms for referencing attachments:

- CID – Content-ID MIME – scheme URL to refer to the attachment that has a CID header with a value corresponding to the URL;
- Content-Location MIME – a URL to refer to an attachments containing Content-Location MIME header.

WS-Security limits attachments referencing to CID scheme URLs.

In order to protect the message against attachment insertion or removal, all attachments should be included in a signature calculation. When an unauthorized attachment insertion takes place (in transit), then the reception of any unsigned attachments is an inevitable indication such. In case of an unauthorized attachment removal, the receiver fails on verifying the original signature.

MIME part reference transforms definition

The MIME transforms definition [21] allows clear and explicit statement of what is included in a MIME reference.

- Attachment-Content-Signature-Transform – indicates that only the content of a MIME part is referenced for signing. The content of the MIME part should be canonicalized.
Attachment-Complete-Signature-Transform – the content and selected headers of the MIME part are referenced for signing. Possible MIME headers to be included are:

- Content-Description
- Content-Disposition
- Content-ID
- Content-Location
- Content-Type

Attachment-Ciphertext-Transform – only the content of a MIME part is referenced, and contains the ciphertext related to an XML EncryptedData element.

Encryption

Encryption may protect the MIME part content. This is done using XML Encryption to encrypt the attachment, placing the resulting cipher text in the updated attachment body replacing the original content, and placing a new <xenc:EncryptedData> element in the <wsse:Security> header.

Example of a SOAP attachment

```
Content-Type: multipart/related; boundary="BoundaryStr" type="text/xml"
--BoundaryStr
Content-Type: text/xml
<S11:Envelope
 xmlns:S11="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
 wsswssecurity-secext-1.0.xsd"
 xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
 xmlns:d="http://www.w3.org/2000/09/xmldsig#">
 <S11:Header>
  <wsse:Security>
   <wsse:BinarySecurityToken wsu:Id="Acert"
    EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
    wss-soap-message-security-1.0#Base64Binary"
    ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
    wss-x509-token-profile-1.0#x509v3">
    ...
   </wsse:BinarySecurityToken>
   <xenc:EncryptedKey Id="EK">
    <EncryptionMethod Algorithm="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
     wss-soap-message-security-1.0#RSAEncryption"
     KeyInfoId="keyinfo">
     <wsse:SecurityTokenReference>
      <ds:KeyInfo Id="keyinfo">
       <ds:X509IssuerSerial>
        <ds:X509IssuerName>
         DC=ACMECorp, DC=com
        </ds:X509IssuerName>
        <ds:X509SerialNumber>12345678</ds:X509SerialNumber>
       </ds:X509IssuerSerial>
       <ds:X509IssuerSerial>
        <ds:X509IssuerName>
         DC=ACMECorp, DC=com
        </ds:X509IssuerName>
        <ds:X509SerialNumber>12345678</ds:X509SerialNumber>
       </ds:X509IssuerSerial>
      </ds:X509IssuerSerial>
     </wsse:SecurityTokenReference>
     <ds:X509Data>
      <ds:X509IssuerSerial>
       <ds:X509IssuerName>
        DC=ACMECorp, DC=com
       </ds:X509IssuerName>
       <ds:X509SerialNumber>12345678</ds:X509SerialNumber>
      </ds:X509IssuerSerial>
     </ds:X509Data>
    </xenc:EncryptedKey>
   </wsse:SecurityTokenReference>
  </ds:KeyInfo>
  <CipherData>
   <CipherValue>xyzabc</CipherValue>
  </CipherData>
  <ReferenceList>
   <DataReference URI="#EA"/>
   <DataReference URI="#ED"/>
  </ReferenceList>
 </S11:Header>
</S11:Envelope>--BoundaryStr--
Additional remarks

When using both signature and encryption, there is a confusion as to whether encrypted content need first be decrypted before signature verification. This confusion can occur if the order of operations is not clear. This problem may be avoided with SOAP Message Security for SwA when attachments and corresponding signatures and encryptions are targeted for a single SOAP recipient (one SOAP header for every recipient).

There is no common agreement about MIME being the right choice for marriage with WS-Security. Recipient of a SwA message needs to scan through the message body for the MIME boundary markers in order to determine where the attached part ends. It will be obviously much simpler if the length would be known in advance. An alternate attachment scheme known as Direct Internet Message Encapsulation (DIME) attempted to address this issue. Another competitor is Message Transmission Optimization Mechanism (MTOM). It differs from SwA in way of representing attachments. When the receiver processes SOAP message, MTOM incorporates the attachments inside the appropriate locations of the original message, pretending that attachments are present inline in XML. Thus, MTOM makes encrypting/signing attachments no different than encrypting/signing SOAP messages without attachments.
4.3 WS-I Basic Security Profile

WS-I Basic Security Profile [32] presents a set of non-proprietary specifications which can be used as components of WS applications. This standard contains also some explanations and adjustments to specifications presented above. Several additional security considerations and recommendations are also addressed.

Creation of this profile was driven by set of guidelines. The essential ones are outlined below:

- **Interoperability.** The profile is an afford for extending interoperability by addressing some common problems which have already revealed. Any inconsistencies and design flaws in presented specifications are addressed only if interoperability issues could be affected. The profile concerns interoperability at the web services layer only. It utilize an assumption that lower-layer interoperability is adequate and well-understood.

- **Focus profiling effort.** The profile contain many different specifications. Some of them are presented in-detail and other are only outlined in minimal way necessary to allow meaningful explanations for in-scope subjects.

- **Testability.** When it is possible this standard presents statements which can be tested in non-intrusive manner. Testing is optional.

- **Multiple mechanisms.** Some of the referenced specifications can contain more then one mechanism which can be used to achieve the same purpose. This profile selects only those of them which are well-understood, widely implemented and useful. Other mechanisms which are extraneous or underspecified are skipped due to interoperability issues.

- **Compatibility with deployed services.** Backwards compatibility with deployed Web services does not concern this profile. This profile does not introduce changes to referenced specifications unless interoperability issues are involved.

- **Do not harm.** The profile has tried to take necessary steps to avoid new security flaws creation.

- **Best practices.** The profile’s intention is not introduce best practices in Web services technologies. If multiple options exists which can lead to any security weaknesses the profile suggests reducing the choice.

**Transport Layer Mechanisms**

The profile points to several cryptographical technologies as:

- HTTP over SSL/TLS
- The TLS Protocol Version 1.0
- The SSL Protocol Version 3.0

The SSL Protocol Version 2.0 is prohibited by the WS-I Basic Security Profile due to known security leaks. The profile specifies mandatory, recommended and discouraged ciphersuites, e.g. cipher suites that use MD5 should be avoided because of reported security weaknesses of this function.
The profile provides guidelines how to process `<wsse:Security>` headers. SOAP messages can be signed and/or encrypted, security tokens can be also incorporated. Guidelines address such problems as order in which of signed or encrypted elements appear in SOAP messages. Both participants of communication must agree which elements are present in messages and which are signed and/or encrypted. This agreement can be done in an out of band fashion. The profile is flexible here and allows addressing this problem differently by each vendor.

The profile specifies that the envelope, header and body elements must not be encrypted. Encrypting these elements would break the SOAP processing model.

**Security Headers**

**Timestamp**

The profile defines an element to store timestamp data, with the following constraints on it:

- Only one `TIMESTAMP` element can be included in security header.
- Only one element called `CREATED` can be included in `TIMESTAMP` element.
- No more than one element called `EXPIRES`(can be optional) in security header.
- Element `CREATED` must precedes `EXPIRES`.

An example of well defined `TIMESTAMP` element is presented below:

```xml
<wsu:Timestamp wsu:Id="timestamp">
  <wsu:Created>2001-09-13T08:42:00Z</wsu:Created>
  <wsu:Expires>2001-10-13T09:00:00Z</wsu:Expires>
</wsu:Timestamp>
```

**Signing and encryption**

XML Signature is used for securing SOAP messages. The profile defines some constraints on XML Signature according to SOAP Message Security:

- Enveloping signatures must not be used.
- The Profile encourage to use detached signatures than including signatures into source message.
- Preferred hash function is SHA-1.

XML Encryption is used for securing SOAP messages. The profile defines also a few constraints on XML Encryption. Most notably encrypted key must precede encrypted data inside security header.
Binary Security Tokens

Base64Binary encoding is used for SOAP message security. If BinarySecurityToken element is used, encoding-type element must be used as well. Incorrect and correct use of BinarySecurityToken is presented below.

- Incorrect:

```
<!-- This example is incorrect because the wsse:BinarySecurityToken element is missing an EncodingType attribute -->
<wsse:BinarySecurityToken wsu:Id='SomeCert' 
    ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0##X509v3">
    lui+Jy4WYKgJW5xM3ahLxOpGVlpzSp4V486hHFe7sHET/uxxVBov7TJ1A2RnW5WkXm9jAEdslm/...
</wsse:BinarySecurityToken>
```

- Correct:

```
<wsse:BinarySecurityToken wsu:Id='SomeCert'
    ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0##X509v3"
    EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0#Base64Binary">
    lui+Jy4WYKgJW5xM3ahLxOpGVlpzSp4V486hHFe7sHET/uxxVBov7TJ1A2RnW5WkXm9jAEdslm/...
</wsse:BinarySecurityToken>
```

Username Token

Only one password element must be included in a Username Token.

X.509 Certificate Tokens

The profile references the following specifications for using X.509 Certificate Tokens in SOAP messages:

- Web Services Security: X.509 Token Profile 1.0, Errata 1.0 Committee Draft 200512, December 2005 [36].
- RFC2459: Internet X.509 Public Key Infrastructure Certificate and CRL Profile [37].

Kerberos Token

Kerberos can be used as a security mechanism in Web Services Security. The profile follows specification indicated below:

- Web Services Security: Kerberos Token Profile 1.1, OASIS Standard Specification, 1 February 2006 [38]

SAML Token

SAML specification (see section 4.7 on page 66 below) if referenced by the profile:

Attachments Security

For security requirements concerning message attachments, the profile references Attachments Profile Version 1.0 (AP1.0) [21] and SOAP Messages with Attachments (SwA) Profile 1.1 [40].

Additional remarks

The profile specifies a set of security considerations that should be addressed when using technologies outlined in the Basic Security Profile 1.0:

- SOAPAction header: The use of SOAPAction header can cause security risk when the transport layer does not provide the same protection to the SOAPAction header as to message content.
- Clock synchronization: The profile outlines some time-based mechanisms which can operate properly only if time is synchronized.
- Using digest Passwords: The profile suggests using digest password than clear text.
- SHA-1: The profile recommends using SHA-1 as the hash function. Nowadays, SHA-1 is sometimes considered obsolete (similarly to MD5), currently there are, however, stronger hash functions in the SHA family (SHA-256, SHA-512).
- Signing Security Tokens: Since tokens store claims about some system entity they should be protected against falsification.

4.4 WS-Policy / WS-PolicyAttachment

WS-Policy and WS-PolicyAttachment standards define general model and syntax for describing the security policies of the Web Services.

The WS-Policy [33] defines a policy model which can be a collection of policy alternatives, where each policy alternative is a set of policy assertions. Policy assertions can point to different requirements such as authentication scheme, transport protocol selection or QoS rules. The reason of creating WS-Policy is to provide single consistent policy grammar. This standard does not specify any mechanisms which can be used to manage policies among SOA compliant systems, especially how to find or attach policy rules to Web Services applications. However, another specification – WS-PolicyAttachment [34] is intended to specify such mechanisms.

The whole goal of WS-Policy is to offer necessary mechanisms which can be further used for specifying policy information for WS applications. The WS-Policy defines XML entity called policy expression for storing domain-specific policy information and mechanisms for describing combinations of policy assertions applying to web services. WS-Policy information can be embedded in a WSDL service description, thus making it easy to expose service policies through a UDDI registry.
All implementations of this specification must use the XML namespace provided by the URI address:


**WS-Policy model**

**Policy Assertion**

The core elements of policy statements are policy assertions. Policy assertions define a particular requirement or other useful property of web services applications. Each policy assertion can define different property (e.g. some security considerations or transactional processing requirements). Domain-specific assertions should be defined in other specification created especially for describing particular assertions.

Assertions defined by domain authors should be strongly typed. The type is established by XML infoset such as `namespace name` or `local name` properties of the root element information item describing the assertion. This assertions must be consistently interpreted no matter what subject is.

The authors of policy assertions can create more complex assertions with policy expressions inside them as children. Children elements can go deeper in assertion semantics and further qualify one or more particular aspects of the assertion.

The authors of assertions are encouraged by WS-Policy to specify simple assertions as much as possible.

**Policy Alternative**

A policy alternative is a collection of policy assertions. Although the order of processing policy alternatives is not specified by the WS-Policy standard, it can be further defined precisely by other standards. The final aggregate behavior of a policy alternative is indicated by the assertions included in this policy alternative. Mechanisms which describes how to determine aggregate behavior are dependent of assertions types and are outside of WS-Policy.

**Policy**

Policy entity is a collection of policy assertions which can optionally be empty. Policy with any policy alternatives represents a kind of requirements which are encapsulated within a policy alternative and can be chosen. The order of policy alternatives entities within policy entity is irrelevant in this specification but can be further specified in other standards. Empty policy represents no choices for requirements.

Policies are used in web services for describing conditions under which both sides can interact. Typically the service provider exposes a policy which describes the terms under which the service is provided. The other side of communication can read the policy and decide to comply with it or not. The requester can choose any of the policy alternatives provided by the policy. Each policy alternative is valid and authorizes the requester to consume the service.
Policy assertion is supported by the service consumer only if all conditions described by that policy assertion are satisfied. Policy alternative is supported by the service consumer only if all policy assertions included within that policy alternative are satisfied. Finally, a policy is supported by the service consumer only if at least one policy alternative is supported. Obviously more then one policy alternative can be supported by the service consumer at the same time. Policy can deliver backward-compatibility through policy alternatives versioning. It is possible that the service consumer does not understand all policy assertions. This is useful when delivering new versions of policy assertions which are not understood by all service consumer yet.

**Policy Expression**

**Normal form of policy expression**

Policy entity is represented as a XML infoset. *Normal form* of the policy is a straightforward XML entity with each policy alternative entities being enumerated. Further, each policy assertions are within policy alternative are also enumerated. Normal form of representation is preferred but more compact form is also allowed. The schema of the policy in the normal form is outlined below:

```
<wsp:Policy ...
  <wsp:ExactlyOne>
    <wsp:All> ( <wsp:All> ( <Assertion ...> ... </Assertion> )* </wsp:All> )* </wsp:ExactlyOne>
</wsp:Policy>
```

A policy entity is represented by an XML element outlined below:

```
/wsp:Policy
```

A collection of policy alternatives is represented by an XML element outlined below:

```
/wsp:Policy/wsp:ExactlyOne
```

A policy alternative is represented by an XML element outlined below:

```
/wsp:Policy/wsp:ExactlyOne/wsp:All
```

A policy assertion is represented by an XML element outlined below:

```
/wsp:Policy/wsp:ExactlyOne/wsp:All/*
```

If an policy assertion contains a nested policy element, that element must have at most one policy alternative. A more detailed policy entity is presented below:

```
(01) <wsp:Policy
    xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy"
(02)   <wsp:ExactlyOne>
(03)     <wsp:All>
(04)       <sp:Basic256Rsa15 />
(05)     </wsp:All>
```

```
</wsp:ExactlyOne>
</wsp:Policy>
```
Line numbered (3-5) and (6-8) present two policy alternatives.

**Policy identification**

An URI address can be associated with policy expression. This outlined below:

```xml
<wsp:Policy ( Name="xs:anyURI" )?
    ( wsu:Id="xs:ID" )?
    ... >
    ...
</wsp:Policy>
```

Each XML entity is described below:

```xml
<wsp:Policy/@Name

An absolute URI address can be used for policy expression identification. WS-PolicyAttachment [34] can be used for referencing to a policy entity in other XML documents by following the URI address.

```xml
<wsp:Policy/@wsu:id

Referencing to policy expression can be done by ID also. More detailed policies entities are outlined below:

```xml
(01) <wsp:Policy
    Name="http://fabrikam123.example.com/policies/P1"
(02)   <!-- Details omitted for readability -->
(03) </wsp:Policy>
```

URI-reference “P1” is associated with policy expression:

```xml
(01) <wsp:Policy
    wsu:Id="P1"
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd" >
(02)   <!-- Details omitted for readability -->
(03) </wsp:Policy>
```

**Compact form of policy expression**

It is possible to define compact form of policy expression. Three more elements are defined to build compact form. The elements are: an attribute to decorate an assertion, semantics for recursively nested policy operators, and a policy reference/inclusion mechanism.
Some assertions can be optional. If some assertion should be optional, another attribute must be used to indicate the optionality. The schema below outlines this attribute:

```xml
<Assertion (wsp:Optional="xs:boolean")? …> … </Assertion>
```

The attribute indicating optionality can be omitted. This is equivalent to setting its value to false. False value should not be set up in the attribute but policy parsers must accept optionality attribute with a value of false. Some example of using optionality attribute is presented below:

```
(01) <wsp:Policy
    xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy"
(02)   <sp:IncludeTimestamp wsp:Optional="true" />
(03) </wsp:Policy>
```

this is equivalent to the following normal form of policy expression:

```
(01) <wsp:Policy
    xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy"
(02)   <wsp:ExactlyOne>
(03)     <wsp:All>
(04)       <sp:IncludeTimestamp />
(05)     </wsp:All>
(06)     <wsp:All />
(07)   </wsp:ExactlyOne>
(08) </wsp:Policy>
```

The `@wsp:Optional` attribute in Line (02) of the first policy example above indicates that the assertion in Line (02) is to be included in a policy alternative whilst excluded from another; it is included in Lines (03-05) and excluded in Line (06). Note that `@wsp:Optional` does not appear in the normal form of a policy expression.

A policy assertion can contain nested policy expression. The schema outlining nesting is presented below:

```xml
<Assertion …>
   …
   ( <wsp:Policy …> … </wsp:Policy> )?
   …
</Assertion>
```

Each element is described as:

```xml
/Assertion/wsp:Policy
```

This element represents a nested policy expression. If there is no `wsp:Policy` element within an assertion, the assertion does not have nested policy expression. Recursive normalizing process is applied to each policy assertions with nested policy expressions. The WS-Policy specifies an algorithm which can be used to normalize complex assertions storing policy expressions.
An example of the compact form of nested policy expression is outlined below:

```xml
(01) <wsp:Policy
    xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy"
(02)   <sp:TransportBinding>
(03)     <wsp:Policy>
(04)       <sp:AlgorithmSuite>
(05)         <wsp:Policy>
(06)           <wsp:ExactlyOne>
(07)             <sp:Basic256Rsa15 />
(08)             <sp:TripleDesRsa15 />
(09)           </wsp:ExactlyOne>
(10)         </wsp:Policy>
(11)       </sp:AlgorithmSuite>
(12)     </wsp:Policy>
(13)   </wsp:TransportBinding>
(14) </wsp:Policy>
```

The normal form of the above example is presented below:

```xml
(01) <wsp:Policy
    xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy"
(02)   <wsp:ExactlyOne>
(03)     <wsp:All>
(04)       <sp:TransportBinding>
(05)         <wsp:Policy>
(06)           <sp:AlgorithmSuite>
(07)             <wsp:Policy>
(08)               <sp:Basic256Rsa15 />
(09)           </wsp:Policy>
(10)       </sp:AlgorithmSuite>
(11)     </wsp:Policy>
(12)   </wsp:All>
(13)     <sp:TransportToken>
(14)         <wsp:Policy>
(15)           <sp:HttpsToken RequireClientCertificate="false" />
(16)         </wsp:Policy>
(17)   </sp:TransportToken>
(18) </wsp:All>
(19)   </wsp:ExactlyOne>
(20)     <sp:TransportBinding>
(21)         <wsp:Policy>
(22)           <sp:AlgorithmSuite>
(23)             <sp:Policy>
(24)               <sp:TripleDesRsa15 />
(25)             </wsp:Policy>
(26)           </sp:AlgorithmSuite>
(27)         </wsp:Policy>
(28)     </sp:TransportToken>
(29)         <sp:HttpsToken RequireClientCertificate="false" />
(30)     </wsp:Policy>
(31)   </sp:TransportToken>
(32) </wsp:Policy>
(33) </wsp:TransportBinding>
```
Policy operators

The WS-Policy standard defines also set of policy operators which can be used for expressing complex policies in normal form. Policy operators can be recursively nested. Since policy operators play an essential role in expressing policy definitions, some examples are presented below.

Empty policy:
- <wsp:All /> expresses a policy with no policy assertions.
- <wsp:ExactlyOne /> expresses a policy with no policy alternatives.

Commutative:

\[
\begin{align*}
\text{Empty policy:} \\
\text{Commutative:} & \quad \langle \text{wsp:All} \rangle \langle !- \text{ assertion 1 } \rightarrow \rangle \langle !- \text{ assertion 2 } \rightarrow \rangle \langle \text{wsp:All} \rangle \\
& \quad \text{is equivalent to:} \\
& \quad \langle \text{wsp:All} \rangle \langle \text{wsp:ExactlyOne} \rangle \langle !- \text{ assertion 2 } \rightarrow \rangle \langle !- \text{ assertion 1 } \rightarrow \rangle \langle \text{wsp:All} \rangle
\end{align*}
\]

and:

\[
\begin{align*}
\text{Commutative:} & \quad \langle \text{wsp:ExactlyOne} \rangle \langle !- \text{ assertion 1 } \rightarrow \rangle \langle !- \text{ assertion 2 } \rightarrow \rangle \\
& \quad \langle \text{wsp:ExactlyOne} \rangle \\
& \quad \text{is equivalent to:} \\
& \quad \langle \text{wsp:ExactlyOne} \rangle \langle !- \text{ assertion 2 } \rightarrow \rangle \langle !- \text{ assertion 1 } \rightarrow \rangle
\end{align*}
\]

Associative:

\[
\begin{align*}
\text{Associative:} & \quad \langle \text{wsp:All} \rangle \langle !- \text{ assertion 1 } \rightarrow \rangle \langle \text{wsp:All} \rangle \langle !- \text{ assertion 2 } \rightarrow \rangle \langle \text{wsp:All} \rangle \\
& \quad \text{is equivalent to:} \\
& \quad \langle \text{wsp:All} \rangle \langle !- \text{ assertion 1 } \rightarrow \rangle \langle \text{wsp:All} \rangle \langle !- \text{ assertion 2 } \rightarrow \rangle \\
& \quad \text{and:} \\
& \quad \langle \text{wsp:ExactlyOne} \rangle \langle !- \text{ assertion 2 } \rightarrow \rangle \langle !- \text{ assertion 1 } \rightarrow \rangle \langle \text{wsp:ExactlyOne} \rangle
\end{align*}
\]
is equivalent to:

```
<wsp:ExactlyOne>
  <!-- assertion 1 --> <!-- assertion 2 -->
</wsp:ExactlyOne>
```

Idempotent:

`wsp:All` and `wsp:ExactlyOne` are idempotent. An example below:

```
<wsp:All>
  <wsp:All> <!-- assertion 1 --> <!-- assertion 2 --> </wsp:All>
</wsp:All>
```

is equivalent to:

```
<wsp:All> <!-- assertion 1 --> <!-- assertion 2 --> </wsp:All>
```

and:

```
<wsp:ExactlyOne>
  <!-- assertion 1 --> <!-- assertion 2 -->
</wsp:ExactlyOne>
```

is equivalent to:

```
<wsp:ExactlyOne>
  <!-- assertion 1 --> <!-- assertion 2 -->
</wsp:ExactlyOne>
```

Distributive:

`wsp:All` distributes over `wsp:ExactlyOne`. An example below:

```
<wsp:All>
  <wsp:ExactlyOne>
    <!-- assertion 1 -->
    <!-- assertion 2 -->
  </wsp:ExactlyOne>
</wsp:All>
```

is equivalent to:

```
<wsp:ExactlyOne>
  <!-- assertion 1 -->
  <!-- assertion 2 -->
</wsp:ExactlyOne>
```

Similarly, the following policy:

```
<wsp:All>
  <wsp:ExactlyOne>
    <!-- assertion 1 -->
  </wsp:ExactlyOne>
```

Nowe technologie informacyjne dla elektronicznej gospodarki i społeczeństwa informacyjnego oparte na paradygmacie SOA

TR-ITSOA-OB7-5-PR-08-01-2.2 „Analysis of security policy requirements in SOA-based technologies”

Distributing \texttt{wsp:All} over an empty \texttt{wsp:ExactlyOne} is equivalent to no alternatives. An example is presented below:

\[
\text{<wsp:ExactlyOne>}
\text{<wsp:All>}\text{<! assertion 1 --><! assertion 2 --><! assertion 3 --><! assertion 4 --><wsp:All>}
\text{<wsp:All>}\text{<! assertion 1 --><! assertion 2 --><! assertion 3 --><! assertion 4 --><wsp:All>}
\text{<wsp:All>}\text{<! assertion 2 --><! assertion 3 --><! assertion 4 --><wsp:All>}
\text{<wsp:All>}\text{<! assertion 2 --><! assertion 4 --><wsp:All>}
\text{<wsp:ExactlyOne>}
\]

is equivalent to:

\[
\text{<wsp:ExactlyOne>}
\text{<wsp:All>}
\text{<wsp:ExactlyOne>}
\text{<! assertion 1 --><! assertion 2 --><! assertion 3 --><! assertion 4 --><wsp:All>}
\text{<wsp:ExactlyOne>}
\text{<wsp:All>}
\]

Complex policy example is presented below:

\[
(02) \text{<sp:RequireDerivedKeys wsp:Optional="true" />}
(03) \text{<wsp:ExactlyOne>}
(04) \text{<sp:WssUsernameToken10 />}
(05) \text{<sp:WssUsernameToken11 />}
(06) \text{</wsp:ExactlyOne>}
(07) \text{</wsp:Policy>}
\]

Optionality attribute and distributing operator is applied to the following normalized form of policy:

\[
(02) \text{<wsp:ExactlyOne>}
(03) \text{<wsp:All> <! @wsp:Optional alternative with assertion -->}
(04) \text{<sp:RequireDerivedKeys />}
(05) \text{</wsp:All>}
(06) \text{<wsp:All /> <! @wsp:Optional alternative without -->}
(07) \text{</wsp:ExactlyOne>}
(08) \text{<wsp:ExactlyOne>}
(09) \text{<wsp:All>}
(10) \text{<sp:WssUsernameToken10 />}
(11) \text{</wsp:All>}
\]
Equivalence attribute is used in the example presented below:

```
<wsp:Policy
   xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy"
   <wsp:ExactlyOne>
     <wsp:All>
       <sp:RequireDerivedKeys />
       <sp:WssUsernameToken10 />
     </wsp:All>
     <wsp:All>
       <sp:RequireDerivedKeys />
       <sp:WssUsernameToken11 />
     </wsp:All>
   </wsp:ExactlyOne>
</wsp:Policy>
```

WS-Policy standard provides an option to share assertions among policy expressions. New policy element named `wsp:PolicyReference` can be used anywhere in policy expression where policy assertions are allowed. The schema below presets a new policy element:

```
<wsp:Policy>
   ...
   <wsp:PolicyReference
      URI="xs:anyURI"
      ( Digest="xs:base64Binary" ( DigestAlgorithm="xs:anyURI" ) )? )?
   ...
   ...
</wsp:Policy>
```

Three examples below present in detail how to use `wsp:PolicyReference` in policy expressions:

```
<wsp:Policy
   xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy"
   xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
   wsu:Id="Protection" >
   <sp:EncryptSignature wsp:Optional="true" />
   <sp:ProtectTokens wsp:Optional="true" />
</wsp:Policy>
```
Policy intersection

There are situations when both participants want to limit policy alternatives to those supported by each other (e.g. confronting obligations vs. capabilities). For instance, a service consumer and a service provider want to agree on how to protect a message. Policy intersection mechanism can be used to accomplish this agreement. Policy intersection identifies compatible policy alternatives at both sides.

Intersection is a commutative, associative function that takes two policies and returns a policy. Determining if two policy alternatives are compatible requires domain-specific processing generally. Two policy alternatives must have at least the same vocabulary. The set of rules governs this processing:

- Two policy assertions are compatible if they have the same type.

- If either assertion contains a nested policy expression, the two assertions are compatible if they both have a nested policy expression and the alternative in the nested policy expression of one is compatible with the alternative in the nested policy expression of the other.

- Two policy alternatives are compatible if each assertion in one alternative is compatible with an assertion in the other, and vice-versa. If two alternatives are compatible, their intersection is an alternative containing all of the assertions in both alternatives.

- Two policies are compatible if an alternative in one is compatible with an alternative in the other. If two policies are compatible, their intersection is the set of the intersections between all pairs of compatible alternatives, choosing one alternative from each policy. If two policies are not compatible, their intersection has no policy alternatives.

Examples below present two policies and an intersection of these policies:

```
(01) <wsp:Policy
    xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy"
      <!-- Policy P1 -->
    (02) <wsp:ExactlyOne>
    (03) <wsp:All> <!-- Alternative A1 -->
    (04) <sp:SignedElements>
    (05) <sp:XPath>/S:Envelope/S:Body</sp:XPath>
    (06) </sp:SignedElements>
    (07) <sp:EncryptedElements>
    (08) <sp:XPath>/S:Envelope/S:Body</sp:XPath>
    (09) </sp:EncryptedElements>
    (10) </wsp:All>
    (11) <wsp:All> <!-- Alternative A2 -->

(01) <wsp:PolicyReference URI="#Protection" />
(03) <sp:OnlySignEntireHeadersAndBody />
(04) </wsp:Policy>
(01) <wsp:Policy
    xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy"
      <!-- Policy P2 -->
    (02) <sp:IncludeTimestamp />
    (03) <wsp:PolicyReference URI="#Protection" />
    (04) <sp:OnlySignEntireHeadersAndBody />
    (05) </wsp:Policy>
```
Nowe technologie informacyjne dla elektronicznej gospodarki i społeczeństwa informacyjnego oparte na paradgmacie SOA

TR-ITSOA-OB7-5-PR-08-01-2.2 „Analysis of security policy requirements in SOA-based technologies”

Second policy:

Finally the output policy intersection:
Additional remarks

WS-Policy defines several security considerations about policies, e.g.:

- Policies and assertions should be signed in order to prevent tampering.
- Policies should not be authorized unless being signed.
- Policies should have security token being associated with them. The associated security token specify the signer who is authorize for the scope containing the policy.

4.5 Web Services Security Policy Language (WS-SecurityPolicy)

WS-SecurityPolicy [51] specifies a set of assertions which are used for describing how to secure SOAP messages. This specification introduces a new abstraction – security binding. Security binding is used for grouping individual assertions in a common usage pattern. The bindings describe how the security constrains are actually processed by service participants.

Protection Assertions

Protection assertions are used to identify what parts of request/response messages should be protected and at what level of protection must be adopted. There are two subgroups of this assertion type: integrity assertions and confidentiality assertions.

SignedParts Assertion

SignedParts assertion (belonging to integrity assertions) is used to indicate which parts of message require integrity protection. There can be more than one SignedParts assertion present. A syntax example is presented below:

```xml
<sp:SignedParts ... >
<sp:Body />? <sp:Header Name="xs:NCName”? Namespace="xs:anyURI" ... />* ...
</sp:SignedParts>
```

SignedElement Assertion

SignedElement assertion (belonging to integrity assertions) is used to indicate arbitrary elements for signing in the message. The schema of this assertion is outlined below:

```xml
<sp:SignedElements XPathVersion="xs:anyURI”? ... >
<sp:XPath>xs:string</sp:XPath>+ ...
</sp:SignedElements>
```

EncryptedParts Assertion

EncryptedParts assertion (belonging to confidentiality assertions) is used to indicate which parts of the message require confidentiality. The schema of this assertion is presented below:
EncryptedElements Assertion

EncryptedElements assertion (belonging to confidentiality assertions) is used to indicate arbitrary elements for encrypting in the message. The schema of this assertion is presented below:

```
<sp:EncryptedElements XPathVersion="xs:anyURI"? ... >
  <sp:Xpath>xs:string</sp:Xpath>+
  ...
</sp:EncryptedElements>
```

RequiredElements Assertion

RequiredElements assertion specifies the set of headers elements which must be included in the message. The schema of this assertion is presented below:

```
<sp:RequiredElements XPathVersion="xs:anyURI"? ... >
  <sp:Xpath>xs:string</sp:Xpath>+
  ...
</sp:RequiredElements>
```

Token assertions

Token assertions indicate the type of security tokens used for message protection. Two most representative token assertions are presented below.

UsernameToken Assertion

UsernameToken entity indicates a requirement for using the username token. The schema of this assertion is presented below:

```
<sp:UsernameToken sp:IncludeToken="xs:anyURI"? ... >
  <wsp:Policy>
    ( <sp:WssUsernameToken10 ... /> | <sp:WssUsernameToken11 ... /> ) ?
  ...
  </wsp:Policy> ?
  ...
</sp:UsernameToken>
```

IssuedToken Assertion

IssuedToken token indicate a requirement for an issued token by using WS-Trust [48]. The schema of this assertion is presented below:

```
<sp:IssuedToken sp:IncludeToken="xs:anyURI"? ... >
  <sp:Issuer>wsa:EndpointReferenceType</sp:Issuer>? ...
</sp:IssuedToken>
```
Other types of acceptable token assertions include:

- X509Token Assertion
- KerberosToken Assertion
- SecurityContextToken Assertion
- SecureConversationToken Assertion
- SamlToken Assertion
- RelToken Assertion
- HttpsToken Assertion

**Security Binding Properties**

Security Binding Properties indicate various conditions for the security binding. The properties can be seen as variables which values can be set up dynamically. In WS-SecurityPolicy, assertions are used set up the properties (examples of such assertions are outlined further in this section). Changes in the property value may change the security binding behavior. A list of properties are presented below.

**Algorithm Suite Property**

This property indicates the algorithm suite necessary for cryptographic calculations. It specifies recommended symmetric and asymmetric encryption algorithms, digest functions and allowed key length as well. The value of this property is typically referenced by a security binding indicating how cryptographic calculation must be done.

**Timestamp Property**

This is a Boolean property. The property indicates whether a `wsu:Timestamp` is present in the `wsse:Security` header. Two values are allowed: ‘true’ and ‘false’. ‘True’ means that the timestamp entity is mandatory and must be tamper proof. The ‘false’ value means that the timestamp entity must not be included in the header. Default value is ‘false’.

**Protection order property**

This property indicates the order of integrity and confidentiality checking in case both of them are necessary to perform. Two values are allowed: ‘EncryptBeforeSigning’ and ‘SignBeforeEncrypting’. Default value is ‘SignBeforeEncrypting’.
Signature Protection property

This is a Boolean property. The property indicates whether the signature must be encrypted. If the property is set up to ‘true’ the primary signature must be encrypted, as well as any other signature confirmation elements. Otherwise the primary signature and signature confirmation elements must not be encrypted. Default value is ‘false’.

Token Protection property

This is a Boolean property which indicates whether the signature must cover the token used to generate that signature. If the property is set up to ‘true’ each token used to generate a signature must be covered by the signature. Otherwise the tokens must not be covered by the signature. Default value is ‘false’.

Entire Header and Body Signature property

This is a Boolean property. It indicates whether the signature digest over a SOAP body/header must cover the entire body/header elements. If the value is set up to ‘true’ each digest over body element must cover the entire body element. The same rule applies to the header element. If the value is set up to ‘false’ each signature digest may apply to any descendant of the body or header elements. Default value is ‘false’.

Security Header Layout property

This property indicates layout rules which must be used when adding new entities to the security header. Four values are allowed ‘Strict’, ‘Lax’, LaxTimestampFirst’ and ‘LaxTimestampLast’.

Security Binding Assertions

AlgorithmSuite Assertion

This assertion indicates a requirement for using an algorithm suite defined under AlgorithmSuite Property which was described earlier in this section. The schema of this assertion is presented below:

```
<sp:AlgorithmSuite ... >
<wp:Policy>
  (<sp:Basic256 ... /> |
  <sp:Basic192 ... /> |
  <sp:Basic128 ... /> |
  <sp:TripleDes ... /> |
  <sp:Basic256Rsa15 ... /> |
  <sp:Basic192Rsa15 ... /> |
  <sp:Basic128Rsa15 ... /> |
  <sp:TripleDesRsa15 ... /> |
  <sp:Basic256Sha256 ... /> |
  <sp:Basic192Sha256 ... /> |
  <sp:Basic128Sha256 ... /> |
  <sp:TripleDesSha256 ... /> |
  <sp:Basic256Sha256Rsa15 ... /> |
  <sp:Basic192Sha256Rsa15 ... /> |
  <sp:Basic128Sha256Rsa15 ... /> |
</wp:Policy>
```
```
<sp:TripleDesSha256Rsa15 ... /> |
  ...
<sp:InclusiveC14N ... /> ?
<sp:SOAPNormalization10 ... /> ?
<sp:STRTransform10 ... /> ?
<sp:XPath10 ... /> ?
<sp:XPathFilter20 ... /> ?
...
</wsp:Policy>
...
</sp:AlgorithmSuite>
```

**Layout Assertion**

This assertion indicates a requirement for using a particular security header layout defined under the Security Header Property which was described earlier in this section. The schema of this assertion is presented below:

```
<sp:Layout ... >
  <wsp:Policy>
    <sp:Strict ... /> |
    <sp:Lax ... /> |
    <sp:LaxTsFirst ... /> |
    <sp:LaxTsLast ... /> |
    ...
  </wsp:Policy>
  ...
</sp:Layout>
```

**TransportBinding Assertion**

This assertion is used in situations in which message protection is provided by a different way than using WSS (e.g. by HTTPS). The schema of this assertion is presented below:

```
<sp:TransportBinding ... >
  <wsp:Policy>
    <sp:TransportToken ... >
      <wsp:Policy>
        ...
      </wsp:Policy>
    ...
  </sp:TransportToken>
  <sp:AlgorithmSuite ... >
    ...
  </sp:AlgorithmSuite>
  <sp:Layout ... >
    ...
  </sp:Layout>
  ...
</sp:TransportBinding>
```
SymmetricBinding Assertion

This assertion is used in situation in which message protection is provided by WSS-compliant mechanisms. Two properties are used with this assertion: Encryption Token and Signature Token properties. The schema of this assertion is presented below:

```xml
<sp:SymmetricBinding ... >
  <wsp:Policy>
    <sp:EncryptionToken ... >
      <wsp:Policy> ... </wsp:Policy>
    </sp:EncryptionToken>
    <sp:SignatureToken>
      <wsp:Policy> ... </wsp:Policy>
    </sp:SignatureToken>
    <sp:ProtectionToken ... >
      <wsp:Policy> ... </wsp:Policy>
    </sp:ProtectionToken>
    <sp:AlgorithmSuite ... > ... </sp:AlgorithmSuite>
    <sp:Layout ... > ... </sp:Layout> ?
    <sp:IncludeTimestamp ... /> ?
    <sp:EncryptBeforeSigning ... /> ?
    <sp:EncryptSignature ... /> ?
    <sp:ProtectTokens ... /> ?
    <sp:OnlySignEntireHeadersAndBody ... /> ?
    ... </wsp:Policy>
  </sp:SymmetricBinding>
```

AsymmetricBinding Assertion

This assertion is used in situation in which message protection is provided by means of WSS-compliant mechanisms. Two properties are used with this assertion: Initiator Token and Recipient Token properties. The schema of this assertion is presented below:

```xml
<sp:AsymmetricBinding ... >
  <wsp:Policy>
    <sp:InitiatorToken>
      <wsp:Policy> ... </wsp:Policy>
    </sp:InitiatorToken>
    <sp:RecipientToken>
      <wsp:Policy> ... </wsp:Policy>
    </sp:RecipientToken>
    <sp:AlgorithmSuite ... > ... </sp:AlgorithmSuite>
    <sp:Layout ... > ... </sp:Layout> ?
    <sp:IncludeTimestamp ... /> ?
    <sp:EncryptBeforeSigning ... /> ?
    <sp:EncryptSignature ... /> ?
    <sp:ProtectTokens ... /> ?
    <sp:OnlySignEntireHeadersAndBody ... /> ?
    ... </wsp:Policy>
  </sp:SymmetricBinding>
```
Additional remarks

The WS-SecurityPolicy specification also recommends to follow some additional practices, e.g.:

- Security policies and assertions should be protected against tampering by signing them.
- Security policies should not be accepted and processes unless being signed by the proper signer who has rights to sign a given policy.
- The mechanism used for policy exchange should be protected against the man-in-the-middle downgrade attack.

4.6 Web Services Enhancements (WSE)

WSE [44] is a remarkable effort for implementing the WS-* security specifications, accomplished by Microsoft. WSE is intended for simplifying the usability of security elements defined in WS-* specifications by offering ready-to-use so-called turnkey security scenarios. WSE was developed to achieve the following main goals:

- To aggregate the most often implemented security scenarios and best practices into relatively small number (five actually) of predefined scenarios for message-level security, hereby providing end-to-end secure communication.
- To create an easy to use and intuitive API which can be used to develop secure Web services.
- To combine WS-* protocols and .NET Framework. The idea was to add support for WSE 3.0 in Microsoft Visual Studio 2005 and take advantage of improvements to the .NET Framework 2.0. Message Transmission Optimization Mechanism (MTOM) was also added. MTOM is a W3C recommendation for sending large amounts of data such as document files and images. Microsoft recommends MTOM to (transparently) send binary data as a part of SOAP messages.

Microsoft puts WSE 3.0 on the path to the Indigo framework for building SOA applications [45]. Indigo and ready to use Indigo-like design concepts are supposed to allow the programmers to focus on services orientation rules rather then communication aspects.

Turnkey security scenarios

The idea of creating security scenarios was to find some interoperable usage patterns in securing message-level communication among distributed web service applications. Five security scenarios were proposed. They should be treated as recommendations. Below each security scenario is described along with typical deployment usage.
**UsernameOverTransport**

- In this scenario protection mechanism is located at the transport level (e.g. SSL certificate). Clients are identified via usernames and passwords. This credentials can be stored in Active Directory or SQL Server.
- Typical usage: Known person to a service. Communication across the Internet or Internet to intranet where the applications have limited security infrastructure. Often SSL is used with another turnkey security scenario used inside the firewall, such as Kerberos.

**UsernameOverCertificate**

- X.509 certificate is used for security protection. Clients are identified with usernames and passwords witch can be stored in Active Directory or SQL Server.
- Typical usage: Known person to a service. Communication across the Internet or Internet to intranet where Public Key Infrastructure (PKI) infrastructure is set up.

**AnonymousOverCertificate**

- In this scenario the security protection use X.509 certificate at the server side. Clients are unidentified or anonymous. Any client having server's public key(via server certificate) can communicate securely with the server.
- Typical usage: Unknown Person to Service. Communication across the Internet or Internet to intranet where the applications are smart clients (e.g. Windows Forms) and a Public Key Infrastructure (PKI) infrastructure is established. Since anyone with the server's public certificate can connect to the service, this is limited to either noncritical services or ones where the server's public key is supplied only to a limited set of companies or individuals.

**MutualCertificate**

- Each side exchange X.509 certificates with each other. This can be used to secure the data exchange between all parties.
- Typical usage: Business to Business. Communication across the Internet or Internet to intranet. This scenario can be used for limited peer-to-peer environments where number of actors are not to big.

**Kerberos (Windows)**

- In this scenario one or more Windows Domains and Kerberos mechanism is needed. Applications are located within Windows Domain with security infrastructure provided by Kerberos. Kerberos provides single sign-on and better performance than PKI with X.509 certificates. Kerberos tickets are used for authentication and message protection. Kerberos also supports delegation, which allows a service to execute on behalf of the calling user.
- Typical usage: Intranet environment with Windows Domain and Kerberos infrastructures.
Policy framework

As we has stated in chapter 3, the SOA paradigm requires the security mechanisms not to be hard-wired into applications. In other words, you can write an application that can be launched in different environments and the security elements can be set up in deployment task. This can be done through declarative policy files.

4.7 Security Assertion Markup Language (SAML)

Security Assertion Markup Language (SAML [22]) is used to define XML-based security assertions about principals and transport them possibly in the form of a WSS-compliant security token [23]. SAML Core is referencing to the general syntax and semantics of SAML assertions and the protocols used to request and transmit those assertions from one service participant to another. SAML protocol does not specify how transmit data, it only refers to what is transmitted, i.e. request and response elements.

Additionally, SAML offers simple proprietary SSO protocol for intra- and inter-domain purposes (Web Browser SSO). It also facilitates inter-domain identity management (federated identity) and identity propagation providing a standard representation of a security token for authentication and authorization purposes.

Actually, SAML not only specifies the assertions, but also describes some facilities to operate on them through bindings and profiles. Bindings determine how SAML requests and responses map onto standard messaging or communications protocols (like SOAP). Profiles describe sample use-cases with particular combinations of assertions, protocols and bindings.

SAML assertions

SAML assertions represent statements about a given subject of the request, typically a principal requesting the service. In general, SAML assertions contain information that service providers use to make access control decisions. Three types of statements are provided:

1. Authentication statements – assert that the assertion subject was authenticated by a particular authority (using AuthorityBinding assertion element) with particular means (AuthenticationMethod) at a particular time (AuthenticationInstant). SAML specification identifies several authentication methods (element AuthenticationMethod) including:
   - Password
   - Secure Remote Password
   - Kerberos
   - SSL/TLS certificate-based client authentication
   - X.509 Public Key
   - SPKI Public Key
   - XKMS Public Key
- XML Digital Signature
- Hardware Token

2. Attribute statements – the assertion subject is associated with the supplied sequence of attributes (described with AttributeName and AttributeNamespace);

3. Authorization decision statements – a request to allow the assertion subject to access the specified resource has been granted or not. Three Decision element values are allowed: Permit, Deny and Indeterminate. These days, the authorization service is likely to be a local service of the service provider, although any external authorization service can be used as well.

An authorization decision statement asserts that a subject is permitted to perform action A on resource R given evidence E. The evidence takes the form of an assertion (it may be an attribute assertion and/or an authentication assertion, for instance). The expressiveness of authorization decision statements in SAML is intentionally limited. More advanced use cases are encouraged to use XACML [54] instead (see section 5.2).

A subject can optionally confirm its identity including a subject confirmation evidence directly in a SAML request. For example, a subject may identify itself as the holder of the private key associated with the X.509 certificate contained in the subject confirmation.

SAML assertions can be signed (using XML Signature specification) and encrypted (using XML Encryption).

**SAML protocols**

SAML protocols describe how certain SAML elements (including assertions) are packaged within SAML request and response elements, and gives the processing rules that SAML entities must follow when producing or consuming these elements. SAML requests are sent by service providers to demand the service requester for returning a respective assertion.

The most important type of SAML protocol request is called a query. There are three types of SAML queries:

1. Authentication query
2. Attribute query
3. Authorization decision query

The result of a SAML query is a response containing an authentication/attribute/authorization assertion. Each request and response is identified with a unique RequestID or ResponseID, consequently. Element InResponseTo of a returned response references the corresponding request (is equal to its ResponseID). Each response may contain a list of Status elements with the following values:

- Success – request was processed successfully
- VersionMismatch – incorrect version of the protocol was requested
- Requestor – some arbitrary error in the request was detected
Nowe technologie informacyjne dla elektronicznej gospodarki i społeczeństwa informacyjnego oparte na paradigmatie SOA

TR-ITSOA-OB7-5-PR-08-01-2.2 „Analysis of security policy requirements in SOA-based technologies”

- **Responder** – some arbitrary error on the responder side was encountered when processing the request

Additionally, **SubStatusCode** can be included to give a more detailed information about the problem. Sub status values are pretty self-explanatory: take **RequestVersionDepreciated**, **RequestDenied** or **ResourceNotRecognized** as examples. Optional **StatusMessage** is a human readable text explanation.

Version 2.0 expands the set of SAML protocols considerably with definition of the following ones:

- **Assertion Query and Request Protocol** – requesting existing assertions by reference or querying for assertions by subject and statement type. Examples of request and response messages are provided below:

```xml
<element name="AssertionIDRequest" type="samlp:AssertionIDRequestType"/>
<complexType name="AssertionIDRequestType">
  <complexContent>
    <extension base="samlp:RequestAbstractType">
      <sequence>
        <element ref="saml:AssertionIDRef" maxOccurs="unbounded"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>

<element name="Response" type="samlp:ResponseType"/>
<complexType name="ResponseType">
  <complexContent>
    <extension base="samlp:StatusResponseType">
      <choice minOccurs="0" maxOccurs="unbounded">
        <element ref="saml:Assertion"/>
        <element ref="saml:EncryptedAssertion"/>
      </choice>
    </extension>
  </complexContent>
</complexType>
```

- **Authentication Request Protocol** – it is used by the service consumer to obtain assertions (containing authentication statements which will be used to establish a security context. An **AuthnRequest** message element is sent to a SAML authority to request a **Response** message (including one or more assertions). An example of **AuthnRequest** message is presented below:

```xml
<element name="AuthnRequest" type="samlp:AuthnRequestType"/>
<complexType name="AuthnRequestType">
  <complexContent>
    <extension base="samlp:RequestAbstractType">
      <sequence>
        <element ref="saml:Subject" minOccurs="0"/>
        <element ref="samlp:NameIDPolicy" minOccurs="0"/>
        <element ref="samlp:Conditions" minOccurs="0"/>
        <element ref="samlp:RequestedAuthnContext" minOccurs="0"/>
        <element ref="samlp:Scoping" minOccurs="0"/>
      </sequence>
      <attribute name="ForceAuthn" type="boolean" use="optional"/>
      <attribute name="IsPassive" type="boolean" use="optional"/>
      <attribute name="ProtocolBinding" type="anyURI" use="optional"/>
    </extension>
  </complexContent>
</complexType>
```
• Artifact Resolution Protocol – it is used to transport SAML protocol messages in a SAML binding by reference instead of by value; this mechanism may require additional protection i.e. mutual authentication, integrity protection, confidentiality from the protocol binding. A sample used of ArtifactResolve is presented below:

```
<element name="ArtifactResolve" type="samlp:ArtifactResolveType"/>
<complexType name="ArtifactResolveType">
  <complexContent>
    <extension base="samlp:RequestAbstractType">
      <sequence>
        <element ref="samlp:Artifact"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

And ArtifactResponse:

```
<element name="ArtifactResponse" type="samlp:ArtifactResponseType"/>
<complexType name="ArtifactResponseType">
  <complexContent>
    <extension base="samlp:StatusResponseType">
      <sequence>
        <any namespace="##any" processContents="lax" minOccurs="0"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

• Name Identifier Management Protocol – it is used when an identity provider wants to change the value and/or format of the identifier (referring to the service participant) or to indicate that this name identifier will not be used any more. An example is presented below:

```
<element name="ManageNameIDRequest" type="samlp:ManageNameIDRequestType"/>
<complexType name="ManageNameIDRequestType">
  <complexContent>
    <extension base="samlp:RequestAbstractType">
      <sequence>
        <choice>
          <element ref="saml:NameID"/>
          <element ref="saml:EncryptedID"/>
        </choice>
        <choice>
          <element ref="samlp:NewID"/>
          <element ref="samlp:NewEncryptedID"/>
          <element ref="samlp:Terminate"/>
        </choice>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

<element name="NewID" type="string"/>
Nowe technologie informacyjne dla elektronicznej gospodarki i społeczeństwa informacyjnego oparte na paradygmacie SOA

TR-ITSOA-OB7-5-PR-08-01-2.2 “Analysis of security policy requirements in SOA-based technologies”

• Single Logout Protocol – it is used when a particular session authority will terminate all sessions provided by his near simultaneously; also this mechanism is used when service participant logs out. An example of request and response is presented below:

Response:

• Name Identifier Mapping Protocol – it is used when a service provider wants to communicate with another service provider but it has not an identifier for the principal, it may used identity provider witch share identifiers with both service providers to send a request using this protocol. Sample request and response messages are shown below:

Response:
SAML bindings

A SAML binding is a mapping of a SAML protocol message onto standard messaging formats and/or communications protocols. For example, the SAML SOAP binding specifies how a SAML message is encapsulated in a SOAP envelope.

In the initial SAML specification two bindings have been defined: SOAP protocol and HTTP POST. SAML 2.0 specification defines the following handful of bindings:

- SAML SOAP Binding (based on SOAP 1.1)
- Reverse SOAP (PAOS) Binding
- HTTP Redirect (GET) Binding
- HTTP POST Binding
- HTTP Artifact Binding
- SAML URI Binding

SAML profile

A SAML profile describes in detail how SAML assertions, protocols, and bindings combine to support a defined use-case. SAML 2.0 describes numerous profiles:

- SSO Profiles:
  - Web Browser SSO Profile
  - Enhanced Client or Proxy (ECP) Profile
  - Identity Provider Discovery Profile
  - Single Logout Profile
  - Name Identifier Management Profile
- Artifact Resolution Profile
- Assertion Query/Request Profile
- Name Identifier Mapping Profile
- SAML Attribute Profiles

Example

```xml
<Response
  IssueInstant="2009-04-01T00:46:02Z" Version="2.0" ID="_c7055387-e2927324b306"
  xmlns="urn:oasis:names:tc:SAML:2.0:protocol"
>`
4.8 REST-related standards

In the following section we briefly overview two standards evolved from native WWW technologies. Therefore, we classify them informally as REST-related, since they may appear a natural choice for raw RESTful services (i.e. not necessarily XML-based).

Simple Authentication and Security Layer (SASL)

Simple Authentication and Security Layer (SASL [71]) is a simple framework which provides authentication and communication security for connection-oriented protocols. It allows protocols to reuse existing security mechanisms and to use any new ones without protocol redesigning. SASL framework is hiding the particularities of protocols from mechanisms and the particularities of mechanisms from protocols.

SASL mechanisms provide authentication data integrity and confidentiality for client-server interactions. A server uses SASL to simply inform a client what security mechanisms (as authentication protocols, for instance) are provided to be chosen by the client (with no preference).

SASL is really straightforward and does not incorporate any protection itself. It performs the negotiation of security mechanisms in clear text only, that can be altered by an attacker (downgrade attacks), or lack of re-keying support for cryptographic mechanisms.

Simple and Protected Generic Security Service API Negotiation Mechanism (SPNEGO)

Simple and Protected Generic Security Service API Negotiation Mechanism (SPNEGO [72]) can be used by service participants to agree upon one of a security mechanisms available to them.

The SPNEGO negotiation is carried out by the following steps:

- the *initiator* (contrary to SASL, here a service requester can initiate the negotiation) sends a list of security mechanisms, sorted from most preferred,
- the *acceptor* acknowledges one of possible security mechanism from the most preferred,
- if none of the mechanisms is decided on, the acceptor rejects the proposal and closes connection.
All negotiation messages are sent in clear text, but it is possible (optionally) to use message integrity code (MIC) token to verify that the list of proposed mechanisms has not been modified.

Most visible use of SPNEGO is in Microsoft's "HTTP Negotiate" authentication extension. It was first implemented in Internet Explorer and IIS to provide SSO capability, later marketed as Integrated Windows Authentication.

There are a few implementations of SPNEGO available for Apache Tomcat, WebLogic or WebSphere platforms. Some implementations combine SPNEGO and Kerberos authentication (Kerberos tickets), extending SPNEGO/Kerberos support for to Java applications and applets. Furthermore, some support for Web services has been developed, by incorporating SPNEGO/Kerberos tokens in the SOAP header [4].

4.9 CORBA-related standards

The Common Object Request Broker Architecture (CORBA [17]) is a standard model of service oriented distributed computing, preceding the SOA model but still emerging. In this section we investigate only one CORBA standard related to the security of distributed services (i.e. objects in CORBA-compliant vocabulary). Other CORBA specifications concerning any security-relevant issues are outside the scope of this report.

Security Domain Membership Management Service

Security Domain Membership Management Service (SDMM) is a CORBA-compliant standard. The SDMM architecture defines interfaces which will be used for interacting with SDMM mechanisms. A sample CORBA application presented in Figure 15 can use SDMM by registering the implementation of the logic with its Object Adapter (OA). The Security Interceptor is obtaining a list of domains the object is in (2.a. and 2.b.). Then, a CorbaSec (Corba Security [18]) service computes security policy for the object.

![SDMM architecture main parts and their relationship with others](image)

Figure 15. SDMM architecture main parts and their relationship with others [27].

An application which wants to use some objects from Object Adaptor (OA) or Object Request Broker (ORB) sends a request. The Object Security Attribute Manager (OSAM)
encapsulates OSA information for all objects under its own authority. The OA have some OSAM Policy which is simply a reference to an OSAM. All OA which do not have any OSAM Policy cannot serve any objects for any applications.

CORBA Security authorization model needs following information to make access control decision:

1. principal's attributes describing privileges,
2. the operation to be invoked,
3. the object's interface repository identifier which is used along with the operation for determining the required rights for the operation in the context of the object's interface, and
4. the access policy domain the object belongs to.
5. **Policy definition languages**

Quite a lot of research has already aimed at finding a suitable form of expressing a security policy and many security policy languages has been proposed. Some of the languages were created with their own new syntax. Others are based on well defined syntax like XML, supported by a wide set of ready-to-use technologies. Each of them has been designed to fulfill specific requirements, most typically access authorization only. However, defining a policy language for specific objective usually rises difficulties to express other security aspects in that language. Unfortunately, as we will show, SOA requires a multi-purpose policy definition language which should reflect numerous security needs.

A crucial consideration about security policy languages is suitable level of abstraction encoded into the policy language. Some of existing low-level languages are easily transformable to device specific routines but become inadequate for specifying policy rules for the overall SOA system. Others are high-level and transformation to platform specific routines becomes a tedious work.

Here we review very representative examples of security policy languages – mainly XACML, Ponder, and SecPAL, along with instances of *specifications languages* intended to express security concerns at system design phase – UMLsec and secUML. Each of the reviewed languages has its strong and weak points. It is therefore hard to decide which one is better than others. The following subsection defines features that should be required to win the competition, then succeeding part of this chapter gives a brief overview of selected languages and their conformance to the specified requirements.

### 5.1 Requirements

First we extract key features of an ideal policy definition language for SOA-based systems. In order for such a language to be not only adequate for the SOA but also complete, easy to use and manage we distinguish the following requirements:

- **Support for distributed and loosely-coupled system.** This is probably the most obvious requirement for SOA-based system where single entities (services), often distributed across large-scale environments, provide building blocks for quite complex applications.

- **Well defined syntax for expressing security relevant issues and formally verifiable.** When specifying policies for security relevant issues we need to ensure that policy assertions are well defined and allow us to express security constrains according to our expectations. Policies for large-scale distributed systems can be very complex with hardly observable dependencies, incomplete or leading to conflicting decisions. Without automated evaluation of policies it will be hard to decide about policy correctness.

- **Support for automatic policy correctness evaluation.** Security policy framework should provide PAP with tools for automatic correctness checking of defined policy rules.

- **Syntactical simplicity to provide interoperability in large scale heterogeneous environment.** Policy language should minimize the amount of necessary policy
Nowe technologie informacyjne dla elektronicznej gospodarki i społeczeństwa informacyjnego oparte na paradigmacie SOA

TR-ITSOA-OB7-5-PR-08-01-2.2 „Analysis of security policy requirements in SOA-based technologies”

statements in order to keep policy simple and, in result, interoperable in large-scale system.

- Modularity and extensibility. It is hard to design security policy language which is suitable for any kind system and any security requirements. There are many security policy languages which are well suited for specific needs. An ideal security policy language should be modular in order to allow easy extensions for custom needs. Policy language should be able to incorporate new entities relevant to express any incoming security constraints.

- Distributed decision support. Large-scale system e.g. based on SOA can be very complex and any centralized approach for making policy decisions will probably be a bottleneck. Distributed decision mechanism should be used to improve scalability of policy language.

- Support for defining policies for the system as a whole. We need a policy language able to efficiently express security constraints and obligations across an entire SOA-based system. Large scale systems, especially-SOA based, can be composed into a federation. In this context we need the ability to express security policies for other autonomic systems, acquire and incorporate security policies from other federated systems, define distributed trust relationships and privilege delegation from one principal to another.

- Multilayer design. Deciding about which policy is applicable to a given target (as service or resource) is non-trivial in a loosely-coupled system, where targets are distributed and often governed by distinct policies (or policy sets). Distributed policies should be organized in a hierarchical manner (similar to object group hierarchies in object-oriented distributed systems like CORBA). Higher layer of such a policy composition should aggregate lower layer rules. The hierarchical approach allows to define base policies near the top (hierarchy root) and fine-tune the policy rules for actual targets on bottom layers. As the composition of distributed policies is very error-prone, the superior layer of the composition could include a metapolicy intended to govern the entire composition and resolve possible conflicts.

- Efficiency. At the bottom layer, the policy language should be transformable to platform specific rules/routines which can be processed efficiently on specific platform.

- Low overhead for policy evaluation. Security policy statements govern many security decisions. Large number of policy statements and complicated decision chains will cause noticeable system slow down. Using native and platform optimized security components is necessary to soften overhead of consulting policy statements. Process of applying security policy into the system should also be optimized to mitigate system slow down.

- Secure discovery mechanism to govern autonomous publication and discovery of a service. Currently a manual discovery is often used, where a human judges of whether to engage a discovered service. With autonomous discovery, a machine makes this decision. Since people usually do not trust a machine to make such significant decisions that could put them at risk, autonomous discovery is often limited to only private or preselected services that have been deemed trustworthy. In true SOA systems, such a limitation is not acceptable.

- Support for information flow control. When a service receives some restricted (confidential) information from the service requester (as the authentication credentials), it should be ensured that the restricted use of this information (e.g. resulting from authority delegation) is not compromised by the service itself, its environment (e.g.
data storage) or any part of the data-flow chain (e.g. services involved by nested invocations).

5.2 Extensible Access Control Markup Language (XACML)

XACML (eXtensible Access Control Markup Language, [54]) is a declarative language based on XML created for managing access to resources. It describes both access control policy and format of request / response used in access control queries. XACML is created by OASIS and its current version 2.0 is recommended by W3C consortium.

**Data-flow model**

Communication between individual entities in XACML data-flow is shown in Figure 16. It conforms almost strictly to the model defined in section 3.1.

![Figure 16. Data-flow model and XACML entities.](image)

First, an access control policy must be created. It involves PAP (*Policy Administration Point*) which creates a policy or policy set. Next, created policies are shared with PDP
(Policy Decision Point) which evaluates the policy and renders an authorization decision. PEP (Policy Enforcement Point) is responsible for passing requests to the PDP entity and executing authorization decisions. In the next step, the PEP sends a request for access to the context handler in its native request format. The context handler entity is responsible for converting decision request in the native request format to the XACML format and converting authorization decisions from XACML to the native response format.

Context handler entity is also responsible for getting all necessary attributes needed by request from PIP entity (Policy Information Point). Context handler can also include some additional resources in the request, for example: a part of the database with user roles or content of the XML file with resource permissions. When all attributes are collected, the context handler entity sends the attributes to the PDP. The PDP evaluates the policy and returns the response with authorization decision to the context handler.

The context handler translates the response to the native response format and returns the response to the PEP. If there are some obligations within the response, PEP fulfills the obligations. Obligations could be e.g.: sending an e-mail to the administrator and informing him about the access to resources.

**Policy language model**

XACML is based on XML. It has XML Schema definition that describes in details XACML grammar and semantics. Figure 17 shows general syntax of XACML language. Main elements of language are rules, policies and policy sets. Each of main elements has target element that indicates the target of rule, policy or policy set.

The rule element is the basic component of each policy. It consists of: a target, an effect and a condition. The target defines the set of resources, subjects, actions and environment to which the rule is intended to apply. An XACML PDP entity verifies that the matches defines by the target are satisfied by the subject, resource, action and environment attributes in the request context. The effect of the rule describes the decision that is generated by the rule when it evaluates to true. Two values are allowed: "Permit" and "Deny". The rule's condition refines the applicability of the rule beyond the predicates implied by its target. If the rule itself evaluates to true, the rule's condition also must evaluate to true.

The policy element is described by: a target, a rule-combining algorithm identifier, a set of rules and obligations. A policy target function specifies the set of subjects, resources, actions and environments to which it applies. The rule-combining algorithm describes the procedure of merging results of the evaluated rules. The decision given in response context by the PDP is the value of the policy, as defined by the rule-combining algorithm. Next element of policy, rules, has already been described above. Last element of policy, obligations, may be added by the writer of the policy. Obligations are returned to the PEP in the response context and describe actions that must be fulfilled by the PEP in access control procedure.
The policy set element contains a target, a policy-combining algorithm identifier, a set of policies and obligations. The target, policy and obligation components are described above. The policy-combining algorithm describes the procedure of combining results of evaluated policies. The decision given in response context by the PDP is the result of evaluating the policy set, as defined by the policy-combining algorithm.

**Example**

This simple example illustrates basic rights checking and methodology for policy writing. Suppose that some company have access control policy that states:

User "root" has right to execute any action on any resource.
The aim of this example is to show how to build correct policy in XACML, based on given rule.

First let us construct a simple request that arrives to the PDP entity. In natural language it could be:

Jon Smith, logged as root user, request access to the file with user's passwords.

In XACML, such a request will be converted to:

```
(1) <?xml version="1.0" encoding="UTF-8"?>
(2) <Request
(3)   xmlns="urn:oasis:names:tc:xacml:2.0:context:schema:os"
(4)   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
(5)   xsi:schematLocation="urn:oasis:names:tc:xacml:2.0:context:
(6)   schema:os http://docs.oasis-open.org/xacml/access_control-
(7)   xacml-2.0-context-schema-os.xsd">
(8)   <Subject>
(9)     <Attribute AttributeId="firma_users_user_username"
(10)       DataType="http://www.w3.org/2001/XMLSchema#string">
(11)       <AttributeValue>
(12)         root
(13)       </AttributeValue>
(14)     </Attribute>
(15)   </Subject>
(16)   <Resource>
(17)     <Attribute
(18)       AttributeId="urn:oasis:names:tc:xacml:1.0:resource:
(19)       resource-id"
(20)       DataType="http://www.w3.org/2001/XMLSchema#anyURI">
(21)       <AttributeValue>
(22)         file://etc/passwd
(23)       </AttributeValue>
(24)     </Attribute>
(25)   </Resource>
(26)   <Action>
(27)     <Attribute
(28)       AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
(29)       DataType="http://www.w3.org/2001/XMLSchema#string">
(30)       <AttributeValue>
(31)         read
(32)       </AttributeValue>
(33)     </Attribute>
(34)   </Action>
(35) </Request>
```

Line 1. contains standard XML document tag. Line 2. introduces root element of the XML document – «Request». Request element has many attributes describing XML namespace declarations. In lines 8-15 there is «Subject» element that contains informations about subject making the access request. In this example only one information about subject is given – username. The «Resource» element in line 16, contains informations about resource to which subject has requested access. In example subject wants to access file /etc/passwd. The «Action» element describes action that the subject wants to take on given resource. In this example, subject wants to read resource.
Now, let us build a policy that is applicable to the considered request:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Policy
   xmlns="urn:oasis:names:tc:xacml:2.0:policy:schema:os"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schematLocation="urn:oasis:names:tc:xacml:2.0:policy:
   schema:os http://docs.oasis-open.org/xacml/access_control-xacml-
   2.0-policy-schema-os.xsd"
   PolicyID="urn:oasis:names:tc:example:SimplePolicy1"
   RuleCombiningAlgId="identifier:rule-combining-algorithm:deny-
   overrides">
   <Description>
     Przykład polityki kontrol dostępu
   </Description>
   <Target />
   <Rule
     RuleId="urn:oasis:names:tc:xacml:2.0:example:SimpleRule1"
     Effect="Permit">
     <Description>
       Użytkownik o nazwie użytkownika `root` na prawo
       wykonywania dowolnej akcji na dowolnym zasobie.
     </Description>
     <Target>
       <Subjects>
         <Subject>
           <SubjectMatch
             MatchId="urn:oasis:names:tc:xacml:1.0:function:string-
             equal">
             <AttributeValue
               DataType="http://www.w3.org/2001/XMLSchema#string">
               root
             </AttributeValue>
             <SubjectAttributeDesignator
               AttributeId="firma_users_user_username"
               DataType="http://www.w3.org/2001/XMLSchema#string"/>
           </SubjectMatch>
         </Subject>
       </Subjects>
     </Target>
   </Rule>
</Policy>
```

Lines 1-10 contain head of the policy element with XML namespace declarations. Line 8. assigns a name to this policy, which has to be unique for a given PDP. In line 9. an algorithm is specified that will be used to resolve the result of various rules that may be defined in the policy. The “deny-overrides” rule-combining algorithm says that, if any rule evaluates to "Deny", then the policy must return "Deny". If all rules evaluate to "Permit", then the policy must return "Permit".

Lines 11-13 provide optional description to the policy. Line 14. specifies the target of the policy. If the subject, resource and action in a decision request do not match the values specified in the policy target, then the policy does not need to be evaluated. In this example, the target section describes policy that is applicable to any decision request. Lines 15-39
specify the rule. In line 17, there is effect element that describes effect this rule has if the rule evaluates to "True". If the rule is satisfied, it will evaluate to "Permit". If a rule evaluates to "False" then it returns a result of "NotApplicable". Line 22. introduces the target of the rule that describes the decision request to which this rule applies. The rule target is similar to the target of the policy. In lines 25-35, «SubjectMatch» element specifies an exact value for the subject element in decision request that makes this rule applicable. Element «SubjectMatch» specifies a matching function in the «MatchId» attribute. In this example this function checks if subject value from decision request is equal to string value "root". Line 39. closes the rule and line 40. closes the policy. This policy has only one rule, but more complex policies may have any number of rules.

The result of this request submitted to the PDP that executes the policy described above is "Permit". The response submitted by PDP to the context handler looks as follows:

```
(1) <?xml version="1.0" encoding="UTF-8"?>
(2) <Response xmlns="urn:oasis:names:tc:xacml:2.0:context:schema:os"
(3)   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
(4)   xsi:schemaLocation="urn:oasis:names:tc:xacml:2.0:context:
(5)   schema:os
(6)   http://docs.oasis-open.org/xacml/xacml-core-2.0-context-schema-
(7)   os.xsd">
(8)   <Result>
(9)     <Decision>Permit</Decision>
(10) </Result>
(11) </Response>
```

**Additional remarks**

There is privacy policy language having very similar syntax to XACML – Enterprise Privacy Authorization Language (EPAL [69]) – proposed by IBM and submitted to W3C in 2003. EPAL and XACML differ in how the overall result of a policy evaluation is determined, given that various rules could return different individual values. They also differ in how errors and obligations are handled. EPAL functionality may be considered as a subset of XACML features [2], e.g. EPAL does not support nested policies or multiple subject categories, but in return, it offers few characteristics unavailable in XACML – such as conflict avoidance and policy vocabulary containers.

Some implementations of XACML are already available on the market: Sun XACML Engine, HERAS® XACML, XEngine, XACML Enterprise (Googlecode project), XACMLight for Apache Axis2 (Sourceforge open source project), JBoss XACML (LGPL project) or Microsoft XACML.NET, among others.

More detailed description and analysis of XACML is included in parallel report TR-ITSOA-OB7-5-PR-09-02 [74]. Comprehensive comparison of EPAL and XACML may be found in [3].

### 5.3 Ponder

Ponder [60] is a language for specifying security and management policies for distributed systems. It has been developed at Imperial College over number of years. Ponder is a declarative, object-oriented language for specifying different types of policies. It can be used to specify security policies with role-based access control and management policies.
Ponder is a platform-independent language, that maps its records to low level functions in existing underlying platform.

**Policy concept in Ponder**

In Ponder, a policy is a rule that can be used to change the behavior of a system. The basic constructs that describes the policy are: *basic* policies, *composite* policies and *meta* policies. All objects defined in system can be grouped in *domains* or *sub domains*. Additionally, there are defined concepts of *role*, *relationship* and management structure.

The *basic policies* include: *authorization* policies, *obligation* policies, *refrain* policies and *delegation* policies. *Authorization policies* are security policies related to access control. They specify what actions a subject is permitted to do, to a set of target objects. *Obligation policies* specify what activities a subject must do to a set of target objects, and are triggered by events. *Refrain policies* specify what a subject must refrain from doing. *Delegation policies* specify which actions subjects are allowed to delegate to others.

*Composite policies* are used to group a set of related policy specifications in roles, relationships and management structures. Thanks to that, policies can share definitions and declarations in one scope.

*Meta policies* are policies that describe coexistence of other policies. They can be used to specify correctness of policy and conflicts detection.

In Ponder a subjects and targets of policies are specified in *domains*. Domains can be used to partition the objects and group objects to which policies apply. A domain is very similar in concept to a file system directory. Only difference between them is that domain may hold references to any type of object, including a person. Ponder uses path names to identify domains.

In Ponder tasks and responsibilities corresponding to the position in organization are grouped into a roles associated with the position. A role is a set of authorization policies describing the rights and obligations for that position. In organization policies and roles interact each other creating common relationships. Role relationships specify polices about the interaction between roles and policies related to shared object. In Ponder there are also policy called management structures that are used to define roles and relationships within an organization unit.

Figure 18 shows a class diagram for Ponder policies. Classes *Object, BasicPolicy, CompositePolicy, auth* and *deleg* are abstract classes. There is a concrete class for each of the Ponder policies. Users can create instances of concrete classes directly, or use type definitions to create user-defined sub-classes.
Basic, composite and meta policies

There are three types of policies in Ponder: basic policies, composite policies and meta policies. The basic policies are elementary components of each policy in Ponder. The body of a basic policy consists of common elements: a subject, a target and a when-constraint. The subject and the target are specified using domain scope expressions. The when-constraint element can be specified optionally to limit the applicability of the policy.

As it was said before, there are four main types of basic policy: authorization, obligation, refrain and delegation policies. An authorization policy specifies access control. It could be a positive authorization policy (which defines the action that a subject is permitted to perform on a target) or negative authorization policy (i.e. specifies the actions that a subject is forbidden to perform on a target). An example of positive authorization policy is shown below.

```
type auth+ serviceManT(subject s, target t) {  
  action resetSchedule, enable, disable;  
} // serviceManT

inst auth+ brService = serviceManT (brManager/, brServices/);
```

This example shows a positive authorization policy named `serviceManT` that defines actions which a subject of this policy is permitted to perform on a policy target. The last line declares an instance of this policy named `brService` with specifies a concrete subject and target of the policy.

The second type of authorization policy is negative authorization policy. It defines the actions that subjects are forbidden to perform, and restricts access rights for resources. An example of negative authorization policy is shown below.
Nowe technologie informacyjne dla elektronicznej gospodarki i społeczeństwa informacyjnego oparte na paradigmie SOA

TR-ITSOA-OB7-5-PR-08-01-2.2 „Analysis of security policy requirements in SOA-based technologies”

Another type of basic policies are obligation policies. They specify actions that a subject must perform on a set of target objects when a given event occurs. In contrast to other policies, obligation policies are always triggered by events and, unlike authorization policies, they are interpreted by subjects. Obligation actions are specified with the use of concurrency operators indicating whether the actions are to be performed sequentially or in parallel. It is imperative to precisely select the objects in the subject or target domain to which an obligation policy applies. An example of obligation policy is shown below:

```plaintext
type oblig perfIncreaseT (subject s, target<Router> t) {
    on perfDegradation(bw, source);
    do t.bwReserve(bw) -> s.log(bw, source);
} // perfIncreaseT

inst oblig pl = perfIncreaseT(brEngineer, coreRouter/+edgeRouter/);

inst oblig perfIncrease {
    subject s = brEngineer;
    target t = coreRouter/+edgeRouter/;
    on perfDegradation(bw, source);
    do t.bwReserve(bw) -> s.log(bw, source);
} // perfIncrease
```

The policy (perfIncreaseT) is triggered by performance degradation. The subject of the policy invokes action bwReserve(bw) on the target object, followed by action log(bw, source).

Refrain policies define actions that the subject must refrain from performing on target objects. Refrain policies can be used when the subject is permitted to perform the action but is asked to refrain from doing so if particular constraints apply. An example of this type of policy is shown below:

```plaintext
inst refrain politeBehaviour {
    subject HQStaff/;
    target /; // Any target
    action videoconference;
    when Time.day() = "fri";
} // politeBehaviour
```

The policy subject is allowed to delegate actions to other subject. This kind of policies is called in Ponder delegation policies. Delegation policies are aimed at delegating subjects’ rights to servers or third-parties enabling to perform actions on their behalf. A negative delegation policy identifies what delegations are forbidden. A delegation policy must specify: the authorization policy from which the delegated rights are derived, the
subjects that can delegate these access rights (grantors) and the objects to whom the access rights can be delegated (grantees). An example of delegation policy is shown below:

```plaintext
type auth+ serviceManT (subject s, target t) {
    action t.resetSchedule, t.enable, t.disable;
} // serviceManT

type deleg+ sDelegT (serviceManT a) (subject grantor, grantee granteeD) {
    action resetSchedule;
} // sDelegT

inst auth+ serviceMan = serviceManT(brManager/, brServices/);
inst deleg+ sDeleg = sDelegT(serviceMan)
    (brEngineer/+brSys/, brServices/);
inst deleg+ sDeleg2 = sDelegT(sDeleg)
    (brEngineer/, resetAgent/);
```

Ponder allows for grouping a number of related policies specifications into a composite policy. The main composite policy is a group. Group is a syntactic scope used to declare a set of policies and constraints which are grouped together. A group can contain any basic policy or nested group specifications. A specific type of composite policy is a role. A role is a group in which all policies have the same subject. A policy called relationship specifies policies pertaining to some relationship of roles rather that the individual participating roles. The last composite policy, a management structure, defines the configuration of roles and relationships in organization units.

Finally, a meta policy in Ponder specifies constraints over a set of policies, individual policies or policy elements. Meta policy can be used to check if there are any conflicts. In the following example the meta policy checks all pairs of policies in its scope for possible conflicts:

```plaintext
type meta dutyConflictT(act1, act2, tarType) raises conflictSepD(z) {
    [z] = self.policies->select(pa, pb |
        pa.subject->intersection(pb.subject)->notEmpty and
        pa.action->exists(act | act.name = act1) and
        pb.action->exists(act | act.name = act2) and
        pb.target->intersection(pa.target)->oclIsKindOf(tarType));
    z->notEmpty;
} // dutyConflictT

inst meta dc = dutyConflictT("execute", "authorise", "payment");
meta bwDc = dutyConflictT("addBandwidth", "use", "service");

oblig notifyConflict {
    subject policyService;
    on conflictSepD(z);
    do policyService.notify(manager);
} //notifyConflict
```

More detailed description and analysis of Ponder is included in report TR-ITSOA-OB7-5-PR-09-02 [74].
Ponder2

There are some significant differences between Ponder and Ponder2. In contrast to the previous version, Ponder2 has been designed as an extensible framework that can be used at different levels of scale, from small embedded devices to complex Virtual Organizations. Most important differences concern the language syntax, new policy types, conflict resolution and communication issues.

Ponder’s design suffered from dependency on a centralized infrastructure and deployment model based on centralized decision-making. Although some of the underlying concepts bear similarity to the basic constructs of Ponder the entire framework has been re-designed in version 2. Ponder2 includes a new high-level configuration and control language (PonderTalk) and user-extensible managed objects programmed in Java. The main characteristics of Ponder2 include:

- Ponder2 is extensible (it allows to dynamically extend the policy environment with new functionality and the policy interface with new resources),
- Ponder2 is interactive (managers and developers can interact with the policy environment),
- Ponder2 is more scalable (policy must be executable also on such resources as PDAs, mobile phones).

Ponder2 introduces a concept of managed cell that is a set of hardware and software components forming an administrative domain able to function autonomously, and thus capable of self-management. Managed cells can interact with each other through asynchronous events. It is also possible for an object from one instantiation of Ponder2 to use objects from another instantiation. Currently there are two protocols included in Ponder2 that supports communication between instances: Java Remote Method Interaction (RMI) and HTTP/SOAP.

Ponder has defined an object-oriented language for writing system-wide policies. Ponder2 introduces a new language called PonderTalk (based in Smalltalk) to control and interact with self-managed cells. Ponder2 managed objects are very similar to Smalltalk classes, but they are written in Java.

PonderTalk is focused on exchanging messages between managed objects. Its statements generally comprise a reference to some managed object followed by one or more messages to be sent to it. Every policy entity represented in Ponder2 inherits from Managed Object class written in Java, capable of receiving and replaying to PonderTalk messages. PonderTalk is also used to describe authorization rules in Ponder2 Authorization Framework.

Another feature of Ponder2 is the possibility to communicate with external applications. This is completely new quality intended to provide security policy functionalities for applications outside of the Ponder2 framework. Ponder2 may receive information from outside applications in order to make its decisions and communicate those decisions back to those applications. Ponder2 uses PonderTalk managed object to communicate with external sources via RMI.
Ponder2 explicitly distinguishes between inbound and outbound Policy Enforcement Points. In Figure 19 four Policy Enforcement Points are presented: PEP1 and PEP4 for the subject side, PEP2 and PEP3 for the target side.

Ponder2 re-implements and re-design the policy types known from its previous version. The main concept of the new policy types is similar those of Ponder, but there is at least one important distinction. Ponder2 policies are dynamic, i.e. they may change at any time by loading, enabling and disabling them without stopping the system. In Ponder there were four basic types of policies: authorization policies, obligation policies, refrain policies and delegation policies. In Ponder2 there are only two basic types: obligation policies and authorization policies. Obligation policies are Event Condition Action (ECA) rules that must perform certain actions when certain events occur. Authorization policies permit or deny requested actions based upon the source and the target of the action, and the action itself.

In Ponder2, similarly to Ponder, the authorization policies can be positive and negative. When two or more policies of opposite sign apply to the same action conflicts arise. In such situations Ponder2 uses a new conflict resolution strategy that deals with conflicts at runtime. In fact, Ponder did not have any actual mechanisms for resolving conflicts. It only makes use of the meta-policy that can detect conflicts statically (conflicts are detected by static analysis of the policy set). However it is often impossible to perform such an analysis on policies that depend on a run-time state. In order to overcome that problem, Ponder2 offers a strategy to deterministically resolving conflicts between two or more policies. Conflict resolution algorithm is based on domain nesting. The domain nesting resolution determines precedence of policies that apply to an instance of subjects, targets, or both (i.e. a policy that applies to a sub-domain is more specific than the one that applies to any ancestor domain). The main strength of this approach is that it is intuitively applicable to a domain-based environment, like SOA-based systems.
5.4 Security Policy Assertion Language (SecPAL)

SecPAL ([46], [47], [55]) is a logic-based decentralized authorization language. It is a Microsoft Research project intended to provide a good balance between syntactical and semantical simplicity on one hand, and policy expressiveness and execution efficiency on the other.

Main features of SecPAL include: expressiveness, clear, readable syntax, succinct and unambiguous semantics, effective decision procedures, and extensibility. Expressiveness of SecPAL allows to define decentralized authorization policy with delegation of rights, constraints (like separation of duties and threshold, expiration requirements, among others), and optional negation of these. Clear, readable syntax emerges from a very simple structure and statements which are very close to natural language. Succinct, unambiguous semantics follows from very limited set of deduction rules that directly specify meaning of SecPAL assertions. Effective decision procedures is confirmed by translation into Datalog with Constraints [56] with polynomial data complexity. Finally, extensibility is supported with the many predefined extension points, as, for example, possibility to add new constraints to the language.

SecPAL has an advantage of automatic translation into XML syntax, widely used by almost any applications. Every XML assertion may be encoded and signed by its issuer.

Example

Below we give a simple example of three SecPAL assertions:

(1) STS says junior is a staff
(2) unixlab says junior can read file://data
(3) junior says HPC can read file://data/hpc if currentTime() < 1/04/2009

In the above example principals are: junior, unixlab, HPC. Assertion (1) is actually an identity token issued by STS, which is trusted by others servers. This assertion says that user junior is member of group (role) staff. Assertion (2) expresses capability for junior, issued by computer unixlab, to read his files (file://data). Assertion (3) delegates to computer HPC right to access a specific file (file://data/hpc), for a limited period of time (until 1/04/2009); it is issued by user junior to support his request.

Related languages

SecPAL raised from experience of access control language of Cassandra system [64] proposed by Becket and Sewell from University of Cambridge in the context of a British national project for Electronic Health Record (HER [65]). Cassandra, on its hand, is based on OASIS policy language ([66]).
Cassandra

Cassandra access policies are expressed directly in Datalog with Constraints. Cassandra uses roles with parameters in order to determine access rights. Roles are a well-known concept in access control models [63]. Typically, a role reflects a job position. According to that concept access rights are assigned to roles instead of principals (thus roles become policy subjects), and principals are assigned to roles. This reduces the burden of processing multiple similar rules for different principals playing similar roles (having same job positions) in the organization. Cassandra extends this concept to parameterized roles, where parameters allow for more fine-grained specification of role’s access rights.

Cassandra offers four access control operations: performing an action, activating a role, deactivating a role, and requesting a credential. It is also possible to express policy idioms like: role hierarchy, separation of duties, and role delegation.

OASIS

Open Architecture for Securely Interworking Services (OASIS [66]) is a system for security data exchange between services, based on RBAC model. The general idea is credential-based role activation (which involves principals who possess credentials, and side conditions depending on the environment state – similar to session concept in RBAC). Principal credentials might be:

- Prerequisite role – a role that this principal have previously activated.
- Appointments – user are authorized to issue other users with appointment certificate, which may be a prerequisite for activating one or more roles. User who has appointment certificate (issued by us) are able to act as us.
- Environmental constraints – e.g. separation of duties.

OASIS extends the notion of roles into roles with parameters such as:

- time of a role activation
- userid of a file’s owner
- attribute of the object being accessed.

Additional remarks

SecPAL is noticeably very friendly to read, but its generality may impose limitations on its completeness as distributed security policy language, i.e. it may not be possible to express all required constrains of security policy for SOA systems.

Some limitations of this language include possibility to express authorization policy only – there is no possibility to express constraints and obligations for the communication, like confidentiality protection of messages. Fortunately, the concept of authorization is very wide here. It is possible, for example, to express rights to specific files on the service side for a given user (like in the example above).
SecPAL supports automatic transformation to XML. Every assertion is may be XML-encoded and signed by its issuer.

More detailed description and analysis of SecPAL is included in report TR-ITSOA-OB7-5-PR-09-02 [74].

5.5 Security specification languages for system modeling

It is well known that many of security flaws can be fairly easy avoided during software development phase. It means that software developers need some mechanisms suitable for incorporate security relevant constraints into system model. UML (Unified Modeling Language) is used as standard for software development process. Not all developers have deep knowledge about security technologies which can be helpful or even necessary for their software but they all need some way to model security relevant issues during development phase. There are languages based on UML which can be used to incorporate security constraints into the system model. These languages are examples of high level abstraction languages. However, those languages are designed to model system architecture not security rules governing security mechanisms. Here we investigate two examples of such specification languages.

**UMLsec**

UMLsec [58] is an extension of the UML language. It adds the possibility to express security-relevant information within the diagrams of designed system specifications. The idea behind UMLsec was that system developers do not always have deep understanding of security-relevant issues. This knowledge is crucial for proper design systems especially those one which are designed for sensitive information processing. UML language is well known and understood by most of system developers. Adding specific extension to the UML could be a good idea to express and improve understanding of security relevant issues by the developers. This, in fact saves cost and time of creating reliable systems.

The main UMLsec extensions mechanisms are: *stereotypes* and *tagged values*. Stereotypes define new types of modeling elements, by extending the semantics of existing types in the UML metamodel. In UML diagrams the stereotypes are presented in double brackets. A tagged value is a name-value pair associating data with model elements. The tagged values are presented in curly brackets. All introduced extensions mechanism can be collected into a *profile*. The UMLsec uses the following diagram types to describe different views of a system: Use case diagrams, Activity diagrams, Class diagrams, Sequence diagrams, Statechart diagrams and Deployment diagrams. Additionally Security-goal-tree diagrams are used to provide more detailed information in subsequent design phases.
Use case diagrams capture security requirements for the modeled system. An example is outlined below:

A Use case diagram on Figure 20 depicts a simplified buying situation. A business process of buying a good, whatever, is expected to execute in a way that provides a fair trade. The customer receives the good in return of a payment. The actions “buy goods” and “sells goods” should be connected in a sense that if the first action occurs, the other one eventually occurs as well. The “U” in the goal tree means “undetermined”. It is not known yet whether the goal will be satisfied (“S”) or denied (“D”).

Activity diagrams are used to present more detailed information than use case diagrams. Figure 21 presents sample activity diagram related to the previous example of a use case diagram.

The diagram in Figure 21 provides more information about fair exchange procedure. Both parties of fair exchange are divided in a sense the actions they perform. Activities can be easily seen.
SecureUML

SecureUML [59] is another modeling language based on UML. The idea of using UML as a basis for modeling security relevant issues is very similar here to the UMLsec. However, SecureUML is more oriented on modeling access control policies. Access control policies described by SecureUML can be added to the main stream of model-driven software development process, thus improve understanding of security relevant issues by system developers who not always are security experts. SecureUML conforms to the Model Driven Architecture (MDA) – a standard architecture for model-driven software development.

SecureUML uses role-based access control (RBAC) approach as background security model. RBAC model is widely used in complex software system e.g. databases. It decouples users and their privileges into an entity named role which can be seen as a method of gathering and assigning privileges more efficiently than on particular user level.

Similarly to UMLsec, SecureUML also uses UML stereotypes and tagged values. The stereotypes are used to define new types of model elements. The tagged values are used to define additional attributes on metamodel types. Additionally, SecureUML uses Object Constraint Language (OCL) to express authorization constraints.

SecureUML metamodel defined by the authors is presented in Figure 22.

![Figure 22. Metamodel of SecureUML [59]](image)

The entities of RBAC are represented as metamodel types: User, Role and Permission. Additional ResourceSet type indicates a user-defined set of model elements used to define permissions or authorization constraints. A Permission is a relation linking a Role to a ModelElement or a ResourceSet. The semantics of a permission is defined by ActionType elements. Each ActionType is used to classify the permission and represents a class of security relevant operations related to a specific resource (e.g. action like execution needs read permissions). A ResourceType element defines all action types available for specific metamodel type. The set of resource types,
their corresponding action types, semantics of actions types on specific platform, define the resource type model for the platform. An AuthorizationConstraint expresses a precondition imposed on an operation on a specific resource. Violation of such constraint results in denying the particular action.

**Example**

Example of a simple SecureUML model and its description is presented below.

![Figure 23. A calendar application model [59]](image)

Figure 23 shows an UML model of a simple calendar application. The logic of the software is the following:

- A calendar, represented by the Calendar object may store several entries accessed by operations like `createEntry()` or `updateEntry()`.
- Each entry represent an appointment with a start date, end date, location and owner.
- SecureUML elements (e.g. `<<secuml.permission>>`) are used to express access control information dependencies arising between system components (objects).

**Additional remarks**

UMLsec and SecureUML offer a well understood design notation for system developers who not always have deep enough understanding of security relevant issues. The main goal
of these specification languages is to provide high level building blocks for system definition which address security requirements and can be easily applied by system developers. However UMLsec and SecureUML are not suitable for defining security policies because they operate only on system model-level. Therefore, they should be considered as system modeling languages only.
6. Concept and design of a new SOA policy language framework

In this section we outline our proposal for a policy definition language for SOA-based systems, which aims at addressing requirements defined in section 3.

As we have seen from the analysis in sections 4 and 5, it is extremely difficult to fulfill multiple requirements that we have recognized. Although, several WS-* (mainly WS-Security, WS-Policy, SAML) specifications can constitute a remarkable toolset for exchanging policy-related information and enforce the security constrains, they were not intended to provide policy definition mechanisms, that is a domain of policy languages (as XACML, Ponder or SecPAL). However, existing languages do not satisfy all necessary requirements. XACML and SAML are widely used standards supported by OASIS and W3C. Their main problem is very verbose XML notation slowing down the processing, requiring software assisted policy authoring and making assertion analysis difficult. Their limitations include the lack of interaction obligations and capabilities, no delegation support and no conflict avoidance procedures. Ponder, in contrast, is a mature language but with no wide acceptance. It has very compact notation, but still is hardly readable because of highly specialized grammar and syntax. Its main advantage in the SOA-based environments is the support of distributed policy relationship. It has an interesting feature of supporting separate positive and negative policy definitions (e.g. refrain policies or negative delegation). However, Ponder is deficient in expressing interaction obligations and capabilities. SecPAL is remarkably the most friendly to read. Moreover, it is formally verifiable, has an important feature of flexibly expressing delegation of authority and offers automatic transformation to XML. Its ability to express a large set of domain-specific constrains makes it a very complete but complex solution. Limitations include again lack of obligations and capabilities.

As we can see, none of the presented languages fulfill all the requirements recognized in section 5.1 what limits their applicability for SOA systems. Therefore, we propose a solution to overcome these limitations.

In this section, we propose a new language and a fine-grained framework to flexibly manage the security information flow using separate entities with well defined functionality. The framework architecture extends the referential policy framework architecture described in section 3.1. Moreover, we adopt a layered approach to build a policy language model. Each layer focuses on a particular subset of the policy prerequisites established in section 5.1. The overall proposal provides fine-tuned building blocks for a global policy of a SOA-based system.

**SOA policy framework architecture**

In this section we overview a SOA policy framework proposal that extends the basic policy framework architecture defined in section 3.1. Figure 24 presents a model of policy data

---

5 The notion of so called obligations in XACML and Ponder covers only a limited scope of the interaction obligations as defined in section 3.1.
flow in the extended framework architecture. We describe only differences from the basic architecture.

Policy Decision Point (PDP) is a crucial component of the overall policy architecture since it is involved in any access control decision thus potentially invoked at every service request. Optimizations such as caching decisions can significantly reduce the number of evaluations. This motivates the explicit introduction of a PDP cache component, transparent to the Policy Enforcement Points (PEP).

The SOA environment is very likely to be split into several policy domains. It is common also in single administration realms, as it enables logical decomposition of the policy accordingly to software components decomposition or resource management decomposition. Opposite to a monolith-type policy, this distributed policy approach improves flexibility of policy definitions (offering fine-grained policies organized in some hierarchy) and improves policy efficiency (allowing at least some decisions to be made locally with the scope of a single policy domain, without consulting other domains). We express distributed policies with existence of multiple PDP and PAP elements (not included in the figure).

The service-side PEP is responsible for requesting policy decisions from PDP (which actually may come from the PDP cache) related to the access control (policy restrictions). It is additionally charged of enforcing the obligations returned by the PDP. On the other hand, the client-side PEP is responsible for enforcing the capability rules of the according policy, obtaining the principal’s credentials by authenticating the client or retrieving the credentials from the session attributes (if the principal has already been authenticated), and managing the credentials accordingly to a delegation if one is defined.

---

6 we kindly remind the reader of notion of a secure session introduced by WS-Secure Conversation [29]
Policy Information Point (PIP), newly introduced in Figure 24, is used for acquiring additional information, unavailable locally for the PDP (in a local PIB), required to resolve proper decisions. Typically, PIP obtains information from other administration domains. This can be useful in federated systems (or Virtual Organizations) where some part of knowledge needed to take proper local decisions can be distributed across multiple autonomous systems. PIP may be e.g. an external Identity Provider for Federated Identity Management (FIdM) or an external Trusted Authority acting as a source of Federated Access control Management data (FAccM).

Policy Audit Log (PAL) repository, also new in Figure 24, is explicitly distinguished, as it plays an important role in policy management, keeping log trails about policy enforcement events.

In fact, we consider the following entities: PIP, PDP, PDP cache, PIB and PAL to be SOA-compliant services, since they may provide their functionalities through well-defined interfaces. Moreover, we allow for a hierarchical relationship of several PDP services, governing fine-grained system sub-domains and composing a larger SOA environment. Interactions between distinct administration domains (top-level) are supported with the use of PIP entities.

Layered policy model

Since SOA environment usually spreads across multiple administration domains, it requires simple solutions for interoperability reasons. Therefore, we should avoid large number of layers. Our policy language model is decomposed in only four layers of abstraction (Figure 25). At the top of our model, the System Model Layer is intended to specify only basic security prerequisites for system model items (i.e. metapolicy). This layer can benefit from UML-based languages, helpful for system model designers. As the metapolicy issues has been addressed in literature (e.g. UMLsec and SecureUML described in section 5.5), we are not concerned here with this layer. The opposite bottom Physical Layer is device and service implementation-specific, and provides only low level access to different functionalities managed by higher layers (examples of such functionalities are file system access control mechanisms or firewall configurations). Also the Physical Layer is out of our interest at this moment. Therefore, for our concept of a SOA policy framework we focus only on the System Layer and Interaction Layer

![Layered policy language architecture](image)

The System Layer is the accurate place for specifying the whole SOA system policy. We believe that the most adequate solution for that goal is a high-level human-readable
Now we define the construct principles of the proposed new policy language for SOA environments. The main idea is to adopt language structures based on a natural language (similarly to SecPAL, for instance). With the purpose of being simple and easy to manage, we propose for that purpose a constrained natural language with a very closed syntax and restricted keywords. Thus, we omit the heavily complex problem of parsing a natural language. The keyword list is not hardcoded into our language framework. It may be easily extended by a system administrator. These keywords should be understandable for all kind of policy managers.

Policy rules, even well-formed using such a structured natural language, can still be incomplete or conflicting with each other. A tool is needed to transform policy rules into an abstract logical model for further correctness evaluation. However, we do not address the problem of correctness evaluation yet in this report.

The Interaction Layer specifies rules related to particular system components (services) by instantiating general rules inherited from the System Layer into a detailed policy assertions for each component. This layer perceives the SOA system as a collection of concrete single services, and gathers the knowledge about service interactions. On this layer, the SOA policy language may be translated into XACML rules, SAML assertions, WS-Policy/WS-SecurityPolicy tokens or SASL security definitions, or any other kind of security specifications understood by the concrete service implementation (or its application platform).

Figure 26 gives an example of using particular instances of the defined layers in a simplified architecture with a composed service (SOA application). On the System Layer, the SOA policy defines the policy rules for the whole application, which requires nested interactions between several services. The Interaction Layer defines individual rules for each particular service (e.g. XACML policies in this example). Policy data can travel trough the interactions with the use of SAML assertions, for instance.

**General concept of a new SOA policy language**

Now we define the construct principles of the proposed new policy language for SOA environments. The main idea is to adopt language structures based on a natural language (similarly to SecPAL, for instance). With the purpose of being simple and easy to manage, we propose for that purpose a constrained natural language with a very closed syntax and restricted keywords. Thus, we omit the heavily complex problem of parsing a natural language. The keyword list is not hardcoded into our language framework. It may be easily extended by a system administrator. These keywords should be understandable for all kind of policy managers.

Policy rules, even well-formed using such a structured natural language, can still be incomplete or conflicting with each other. A tool is needed to transform policy rules into an abstract logical model for further correctness evaluation. However, we do not address the problem of correctness evaluation yet in this report.

The Interaction Layer specifies rules related to particular system components (services) by instantiating general rules inherited from the System Layer into a detailed policy assertions for each component. This layer perceives the SOA system as a collection of concrete single services, and gathers the knowledge about service interactions. On this layer, the SOA policy language may be translated into XACML rules, SAML assertions, WS-Policy/WS-SecurityPolicy tokens or SASL security definitions, or any other kind of security specifications understood by the concrete service implementation (or its application platform).

Figure 26 gives an example of using particular instances of the defined layers in a simplified architecture with a composed service (SOA application). On the System Layer, the SOA policy defines the policy rules for the whole application, which requires nested interactions between several services. The Interaction Layer defines individual rules for each particular service (e.g. XACML policies in this example). Policy data can travel trough the interactions with the use of SAML assertions, for instance.

**General concept of a new SOA policy language**

Now we define the construct principles of the proposed new policy language for SOA environments. The main idea is to adopt language structures based on a natural language (similarly to SecPAL, for instance). With the purpose of being simple and easy to manage, we propose for that purpose a constrained natural language with a very closed syntax and restricted keywords. Thus, we omit the heavily complex problem of parsing a natural language. The keyword list is not hardcoded into our language framework. It may be easily extended by a system administrator. These keywords should be understandable for all kind of policy managers.

Policy rules, even well-formed using such a structured natural language, can still be incomplete or conflicting with each other. A tool is needed to transform policy rules into an abstract logical model for further correctness evaluation. However, we do not address the problem of correctness evaluation yet in this report.

The Interaction Layer specifies rules related to particular system components (services) by instantiating general rules inherited from the System Layer into a detailed policy assertions for each component. This layer perceives the SOA system as a collection of concrete single services, and gathers the knowledge about service interactions. On this layer, the SOA policy language may be translated into XACML rules, SAML assertions, WS-Policy/WS-SecurityPolicy tokens or SASL security definitions, or any other kind of security specifications understood by the concrete service implementation (or its application platform).

Figure 26 gives an example of using particular instances of the defined layers in a simplified architecture with a composed service (SOA application). On the System Layer, the SOA policy defines the policy rules for the whole application, which requires nested interactions between several services. The Interaction Layer defines individual rules for each particular service (e.g. XACML policies in this example). Policy data can travel trough the interactions with the use of SAML assertions, for instance.
understand and manage, we decide on a *constrained* natural-like language, with only few syntactical constructions allowed.

The constrained natural-like syntax may not necessarily be perfectly convenient for automatic formal verification of correctness of the policy (policy rules). Therefore, a formal representation of the natural-like syntax must also be provided, and the chosen syntax must enable easy transformation to this formal representation.

Hereafter we will define the formal representation of policy rules. Then we will give some exemplary rules to demonstrate the proposed tradeoff between desired syntactical simplicity and required overall functionality of the SOA policy language.

**Formal representation of policy rules**

The formal representation of a policy rule must be general enough to cover all possible aspects of expressing a security policy defined in section 3.1. Namely, it must allow to express restrictions, obligations and capabilities. It must operate on policy entities distinguished by the proposed policy model, i.e. requested *actions*, *subjects* requesting actions, *targets* on which the action is to be executed, and additional conditions applied to such an execution.

First we define a basic form of the formal representation of a policy rule:

```
Rule(Subject, Target, Action, Condition)
```

In this work we propose only a generic structure of the formal expression. The exact instances of formal expressions representing policy rules are out of the scope of current phase of our research and, as such, are not included in this report.

The generic form is sufficiently universal to express all requested functionalities of any SOA-compliant language. Nevertheless, we consider one extension to this basic formal representation of policy rules, aimed at attending exceptional situations occurring possibly in policy processing. An example of such an exceptional situations might be an intrusion detection alert or any similar circumstances transforming the system from its normal runtime to a special state. To preserve the generality of the proposed structure of the formal rule representation, we propose to extend the structure only with an additional global-state condition applied to the rule. This is intended to distinguish normal runtime rules from those that affect the special state of the system.

The complete generic form of the formal representation of a policy rule is therefore the following:

```
Rule(Subject, Target, Action, Condition), Global_condition
```

The formal `Rule` expression returns two possible decision values:

- Allow
- Deny
We also recognize a third decision, which, at least potentially, can be returned by a PDP, in case of any error triggered while processing a policy request:

- **Indeterminate**

Such a decision, if received by a PEP, must be regarded as equal to Deny decision.

### Examples of policy rules

Here we give a few examples of syntactical constructs of the SOA policy language.

#### Restrictions

Generic form of a restriction rule syntax is the following:

```
<Subject> X can access <Target> Y for {<action>}, <condition>.
```

where `<Subject>` is principal (user, role), service; `<Target>` is resource or service; `X` and `Y` are constants or variables representing names, aliases or identities (local or global, including IP address, URL, UDDI); the set `{<action>}` specifies allowed actions related to the target (we consider the following keywords: read, modify, append, create, delete, full access or any access); finally, `<condition>` restricts the actions to a specific time, source, delegation, or any defined predicate (the latter allows for simple policy extensions).

For instance, restriction rules may be:

- **User j_bond can access https://secret/x-file for {read, append}, on Weekdays.**

The above example presented a time-restriction condition. The next one presents location (source) restriction:

- **Role manager can access https://secret/ for {full access}, from local_network.**

**Prohibition rule:**

- **Role manager cannot access https://secret/statistic for {any access}.**

**Delegation:**

- **Service http://secret/x-srv can access https://secret/x-file for {read, append}, on behalf of user j_bond.**

**Predicate condition:**

- **User X can access file://Y for {full access}, if owner(file://Y)=X.**
Predicate conditions can express any desired kind of constraints, also time, location, role etc., as in the following example:

```
User X can access Y for invocation, if Role(X)="secret_agent" and
if Location(Y)="https://secret/service/*" and
if access="GET,POST" and
not on Holidays
```

Generic form of a delegation definition rule syntax is the following:

```
<Subject> X can delegate <Subject> Y for <Target> Z, <condition>.
```

**Obligations**

Generic form of an obligation rule syntax is the following:

```
<Subject> X must <action> with {<attribute>}, <condition>.
```

or

```
<Target> X requires to <action> with {<attribute>}, <condition>.
```

where `<action>` is authenticate, sign or encrypt; and `{<attribute>}` is an ordered set of attribute IDs (e.g. WSS-compliant names of accepted digital signature algorithms).

**Capabilities**

Generic form of a capability rule syntax is the following:

```
<Subject> X can <action> with {<attribute>}, <condition>.
```

or

```
<Target> X can <action> with {<attribute>}, <condition>.
```

Generic form of a distributed trust rule syntax is the following:

```
<Target> X can trust <Subject> Y for {<action>}, <condition>.
```

where `<Subject>` is a principal or an authority.
7. Conclusions

In this report we have given an overview and analysis of various prerequisites for security policy environments compliant to the SOA distributed processing model. Several SOA-based technologies and security-related specifications, standards and recommendations were carefully investigated. Some representative policy languages were evaluated. The presented analysis is necessary for any further development of a new security policy definition language and framework suitable for the SOA and supporting all necessary requirements. We have proposed a preliminary concept of such a policy language model, its syntax and framework.

Forthcoming work will focus on a precise design of the language and its framework, followed be a prototype implementation.
8. Bibliography


Nowe technologie informacyjne dla elektronicznej gospodarki i społeczeństwa informacyjnego oparte na paradygmacie SOA

“Analysis of security policy requirements in SOA-based technologies”


Poznan University of Technology 2009-12-15 106/111


Nowe technologie informacyjne dla elektronicznej gospodarki i społeczeństwa informacyjnego oparte na paradygmacie SOA

TR-ITSOA-OB7-5-PR-08-01-2.2 „Analysis of security policy requirements in SOA-based technologies”


[61] Ponder2 project, http://www.ponder2.net/


9. Main acronyms and dictionary of terms

Main acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>DTD</td>
<td>Document Type Definition</td>
</tr>
<tr>
<td>IP</td>
<td>Identity Provider</td>
</tr>
<tr>
<td>MEPR</td>
<td>Metadata Endpoint Reference</td>
</tr>
<tr>
<td>OA</td>
<td>Object Adaptor</td>
</tr>
<tr>
<td>PAL</td>
<td>Policy Audit Log</td>
</tr>
<tr>
<td>PAP</td>
<td>Policy Administration Point</td>
</tr>
<tr>
<td>PDP</td>
<td>Policy Decision Point</td>
</tr>
<tr>
<td>PEP</td>
<td>Policy Enforcement Point</td>
</tr>
<tr>
<td>PIB</td>
<td>Policy Information Base</td>
</tr>
<tr>
<td>PIP</td>
<td>Policy Information Point</td>
</tr>
<tr>
<td>PKI</td>
<td>Public-Key Infrastructure</td>
</tr>
<tr>
<td>REL</td>
<td>Rights Expression Language</td>
</tr>
<tr>
<td>SAML</td>
<td>Security Assertion Markup Language</td>
</tr>
<tr>
<td>SDMM</td>
<td>Security Domain Membership Management</td>
</tr>
<tr>
<td>SOA</td>
<td>Service Oriented Architecture</td>
</tr>
<tr>
<td>SOAP</td>
<td>formerly Simple Object Access Protocol, lately also Service Oriented Architecture Protocol</td>
</tr>
<tr>
<td>SSO</td>
<td>Single Sign-On</td>
</tr>
<tr>
<td>STS</td>
<td>Security Token Service</td>
</tr>
<tr>
<td>UDDI</td>
<td>Universal Description, Discovery and Integration</td>
</tr>
<tr>
<td>WSDL</td>
<td>Web Services Description Language</td>
</tr>
<tr>
<td>WS</td>
<td>Web Services</td>
</tr>
<tr>
<td>WS-I</td>
<td>Web Services-Interoperability Organization</td>
</tr>
<tr>
<td>WSS</td>
<td>Web Services Security standard</td>
</tr>
<tr>
<td>XACML</td>
<td>eXtensible Access Control Markup Language</td>
</tr>
<tr>
<td>XCBF</td>
<td>XML Common Biometric Format</td>
</tr>
</tbody>
</table>

Dictionary of terms

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>authentication</td>
<td>verification of a principal’s identity</td>
</tr>
<tr>
<td>authorization</td>
<td>control decisions about the access to resources</td>
</tr>
<tr>
<td>attribute service</td>
<td>means a web service designed for maintaining information (attributes) about principals within a trust realm or federation</td>
</tr>
<tr>
<td>canonicalization</td>
<td>conversion from multiple (or ambiguous) forms of some information (like its structure or encoding) into a normalized form of that information</td>
</tr>
<tr>
<td>identity mapping</td>
<td>a method of creating relationships between one or more digital identities or attributes associated by different → identity providers or → service providers with an individual principal</td>
</tr>
<tr>
<td>identity provider</td>
<td>a service that acts as an authentication service to a requestors and a data origin authentication service to service providers; typically an extension of → Security Token Service</td>
</tr>
<tr>
<td>metadata</td>
<td>any data which describes characteristic of a subject; e.g. federation metadata describes attributes in the federation process</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>------</td>
<td>------------</td>
</tr>
<tr>
<td>metadata endpoint reference</td>
<td>location specified as an endpoint reference that allows the requestor to acquire all the necessary metadata for a target service</td>
</tr>
<tr>
<td>nonce</td>
<td>number used once – a random number used to distinguish between invocations or messages (e.g. certificate request) and detect replying participants</td>
</tr>
<tr>
<td>participant</td>
<td>→ service participant</td>
</tr>
<tr>
<td>policy</td>
<td>collection of → policy alternatives</td>
</tr>
<tr>
<td>policy alternative</td>
<td>collection of → policy assertions</td>
</tr>
<tr>
<td>policy assertion</td>
<td>statement representing an individual requirement, capability, or other property of a policy-related behavior in service interaction</td>
</tr>
<tr>
<td>policy assertion parameter</td>
<td>qualifies the policy-related behavior indicated by a → policy assertion</td>
</tr>
<tr>
<td>policy assertion type</td>
<td>represents a class of → policy assertions and implies a schema for the assertion and assertion-specific semantics</td>
</tr>
<tr>
<td>policy attachment</td>
<td>mechanism for associating → policy with one or more policy scopes</td>
</tr>
<tr>
<td>policy expression</td>
<td>formal representation of a → policy in a given language (in WS-Policy standard – XML Infoset, either in a normal form or in an equivalent compact form)</td>
</tr>
<tr>
<td>policy scope</td>
<td>collection of → policy subjects to which a policy may apply</td>
</tr>
<tr>
<td>policy subject</td>
<td>entity (e.g., an endpoint, message, resource, interaction) with which a → policy can be associated</td>
</tr>
<tr>
<td>policy vocabulary</td>
<td>a set of all → policy assertion types used in a given → policy</td>
</tr>
<tr>
<td>principal</td>
<td>any type of entity (e.g. end user or application) that may act as a requestor; a principal is typically represented with a one or more digital identities</td>
</tr>
<tr>
<td>pseudonym service</td>
<td>a service designed for maintaining alternate identity information about principals within a trust realm or federation</td>
</tr>
<tr>
<td>realm</td>
<td>indicate a single unit (domain) of administration or trust</td>
</tr>
<tr>
<td>security context</td>
<td>set of security parameters (e.g. encryption algorithms and session keys) common for a sequence of multiple interactions</td>
</tr>
<tr>
<td>security token</td>
<td>data which confirms various security claims in the service interaction encapsulated in a request/response message</td>
</tr>
<tr>
<td>Security Token Service</td>
<td>a service that provides issuance and management of security tokens</td>
</tr>
<tr>
<td>service</td>
<td>an abstract notion representing a functionality that is accessible to remote clients</td>
</tr>
<tr>
<td>service consumer</td>
<td>client side participant of a client-server interaction</td>
</tr>
<tr>
<td>service participant</td>
<td>any part of a service interaction, either → service consumer or → service provider</td>
</tr>
<tr>
<td>service provider</td>
<td>server side participant of a client-server interaction</td>
</tr>
<tr>
<td>trust</td>
<td>a characteristic that one entity is willing to rely upon a second entity to execute a set of actions and/or to make set of assertions about a set of subjects and/or scopes</td>
</tr>
</tbody>
</table>

* Definition taken from the WS-Policy 1.2 specification [33].