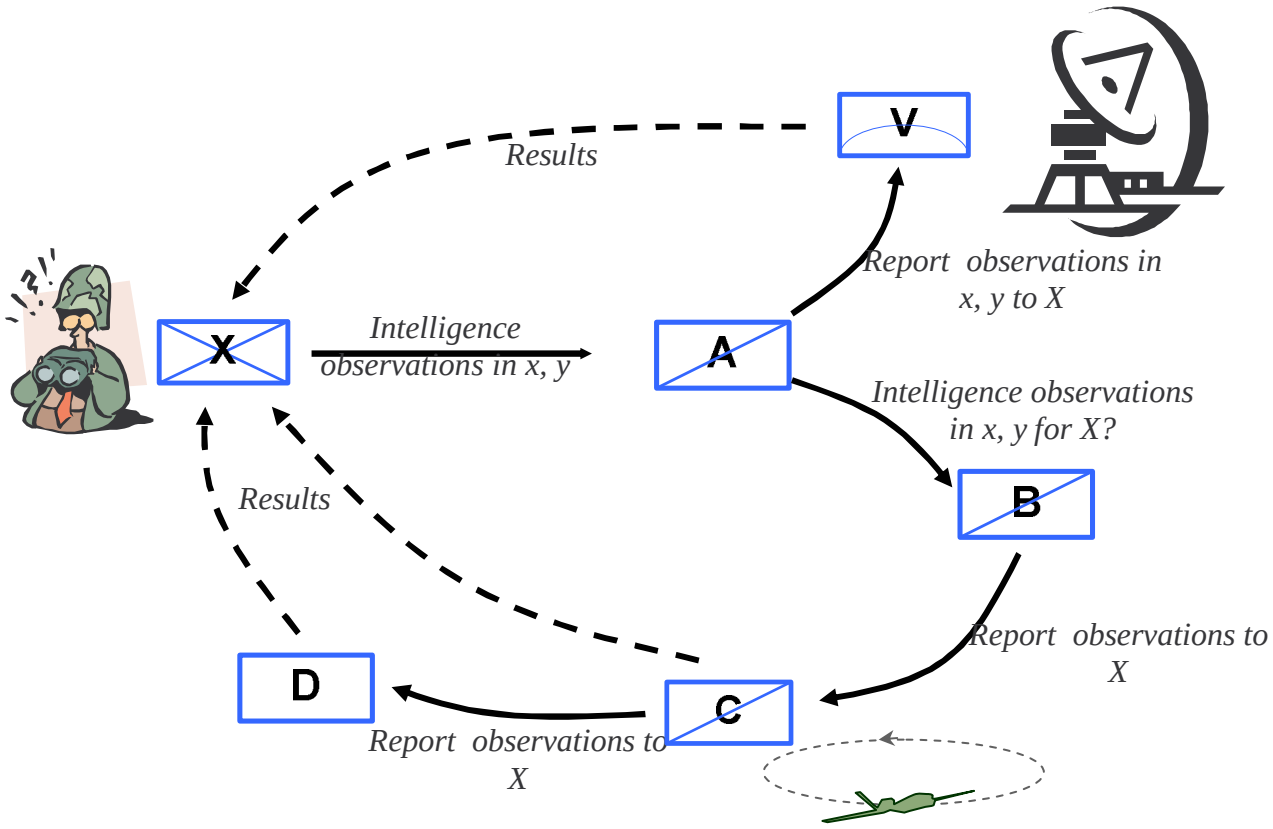
Implementation of Tracing in a Circuit of Web Services

Alf Bengtsson, Dan Nordqvist, Mats Persson,
Lars Westerdahl



FOI-R-- 1792 --SE
ISSN 1650-1942

Command and Control Systems
Scientific report

November 2005

FOI is an assignment-based authority under the Ministry of Defence. The core activities are research, method and technology development, as well as studies for the use of defence and security. The organization employs around 1350 people of whom around 950 are researchers. This makes FOI the largest research institute in Sweden. FOI provides its customers with leading expertise in a large number of fields such as security-policy studies and analyses in defence and security, assessment of different types of threats, systems for control and management of crises, protection against and management of hazardous substances, IT-security and the potential of new sensors.

FOI
Defence Research Agency
Command and Control Systems
P.O. Box 1165
SE-581 11 Linköping

Phone: +46 13 37 80 00
Fax: +46 13 37 81 00

www.foi.se

Implementation of Tracing in a Circuit of Web Services

FOI-R--1792--SE
ISSN 1650-1942

Command and Control Systems
Scientific report

November 2005

Issuing organization	Report number, ISRN	Report type
FOI – Swedish Defence Research Agency Command and Control Systems P.O. Box 1165 SE-581 11 Linköping	FOI-R--1792--SE	Scientific report
	Research area code 4. C4ISTAR	
	Month year November 2005	**Project no.** E7083
	Sub area code 41 C4I	
	Sub area code 2	

Author/s (editor/s)	Project manager
Alf Bengtsson Dan Nordquist Mats Persson Lars Westerdahl	Alf Bengtsson
	Approved by Martin Rantzer
	Sponsoring agency FM
	Scientifically and technically responsible Jonas Hallberg

Report title

Implementation of Tracing in a Circuit of Web Services

Abstract

Web Services is a strong candidate to carry out the Service Oriented Architecture, SOA, which has been established for the future Command and Control System for the Swedish Armed Forces.

A successful progress of the Web Services concept demands flexible ways for Web Services to cooperate and to jointly fulfil a task that is requested by a client. In some applications, the execution of the task is not completely specified beforehand, but could rather be referred to as “best effort”. One example is information searches. To achieve trust in the outcome of the task, it is essential that the identities of the cooperating Web Services can be tracked in a secure way. This report describes an approach to securely track identities of Web Services, subsequently invoked by chains of one-way messages. The model is based on a message structure, which the requesting client can use to iteratively build a hierarchic tree. The model facilitates flexibility and robustness.

The main parts of the model have been implemented, to verify its usefulness. The conclusion is that the model is readily implemented, but that pre fabricated Web Services platforms are not the best choice for implementation.

Keywords

SOA, Web Services, digital signature, tracing, history, security, authentication

Further bibliographic information	Language English
ISSN 1650-1942	**Pages** 51 p.
	Price acc. to pricelist

Utgivare	Rapportnummer, ISRN	Klassificering
FOI - Totalförsvarets forskningsinstitut Ledningssystem Box 1165 581 11 Linköping	FOI-R--1792--SE	Vetenskaplig rapport
	Forskningsområde 4. Ledning, informationsteknik och sensorer	
	Månad, år November 2005	**Projektnummer** E7083
	Delområde 41 Ledning med samband och telekom och IT-system	
	Delområde 2	

Författare/redaktör	Projektledare
Alf Bengtsson Dan Nordquist Mats Persson Lars Westerdahl	Alf Bengtsson
	Godkänd av Martin Rantzer
	Uppdragsgivare/kundbeteckning FM
	Tekniskt och/eller vetenskapligt ansvarig Jonas Hallberg

Rapportens titel

Implementation av spårning i kretsar av samverkande webbtjänster

Sammanfattning

Web Services är en stark kandidat för att realisera den tjänsteorienterade arkitektur, SOA, som har fastställts för det framtida FMLS, Försvarsmaktens Ledningssystem.

En framgångsrik utveckling av konceptet Web Services kräver att det finns flexibla sätt för Web Services att samverka och tillsammans utföra en uppgift som beställts av en klient. I vissa tillämpningar är inte utförandet av uppgiften fullständigt bestämd på förhand, utan den kan i stället betraktas som ett ”bästa försök”. Ett exempel är informationssökning. För att få tilltro till resultatet av uppgiften är det viktigt att identiteterna hos de samverkande tjänsterna kan spåras på ett säkert sätt. I föreliggande rapport introduceras ett sätt att säkert spåra identiteter hos Web Services som anropas via kedjor av enkelriktade meddelanden. Modellen bygger på en meddelandestruktur, som den ursprunglige klienten kan utnyttja till att iterativt bygga upp ett hierarkiskt träd. Modellen resulterar i flexibilitet och robusthet.

För att verifiera modellens användbarhet, har de viktigaste delarna implementerats. Slutsatsen är att modellen är okomplicerad att implementera, men att färdiga plattformar för Web Services inte är det bästa alternativet för implementation.

Nyckelord

SOA. webbtjänster, digital signatur, historik, spårning, säkerhet, autentisering

Övriga bibliografiska uppgifter	Språk Engelska
ISSN 1650-1942	**Antal sidor:** 51 s.
Distribution enligt missiv	**Pris: Enligt prislista**

Contents

Figures

Tables

1 Introduction

This report is part of the research project “Security Aspects within System of Systems”. The project is motivated by a decision made by the Swedish Armed Forces (SWAF) to transform into a more flexible Command & Control (C2) System, based on Service Oriented Architecture (SOA). In this report the term ‘system’ is not referred to as a single system, not even a distributed one, but as two or more cooperating systems with different system owners. In particular, there are requirements on the SWAF C2 system to be able to cooperate with civilian systems as well as with systems belonging to other nations. This report will look into some security issues that arise when tying different systems together.

In [BEN04] a model of cooperating Web Services is described (in Swedish). The services communicate by asynchronously sending one-way messages to each other, thereby forming circuits of cooperating services. Among the characteristics of the model is that the identities of the services involved are part of the messages. They are digitally signed in a way that provides tracing of the identities and that thwarts masquerading and other attacks. The model has been further examined and extended. The report at hand documents the extended model. It also documents an implementation of the main parts of the model. The report is structured as follows.

The report is concluded in section *6 Conclusions*, page 41. It is recommended reading for readers who want an executive summary.

Sections 2-4 are the three main parts of the report. In section *2 Part I: Model and background* the model is described. The description is tied to a military flavoured scenario, which is used throughout the report. The scenario is an information search, which in a natural way is modelled as circuits of Web Services. The client, which requested the search task, can build a state tree from the asynchronously delivered partial results. In this way the client can trace the involved identities. The client can choose to act upon partial results, in a “best effort” way. The hierarchic state tree is the cornerstone of the model. Section 2 is concluded by a discussion of some strengths and weaknesses of the model.

Section *3 Part II: Implementing the trace* is a discussion of an implementation of the two basic parts of the model. Firstly, digital signing/verification of identities and, secondly, building the hierarchic tree from asynchronously received messages. The conclusion of part II is that the model is readily implemented, since it is based upon standard Web Services properties, like XML-messaging and XML Digital Signatures. However, the model is an extension to established standards. A consequence

of the extensions was that two pre made platforms for Web Services were found unsuitable for the implementation, and the implementation was based on standard Java libraries.

Section *4 Part III: Additional security aspects* discusses two security aspects other than the identity verification. It discusses looping, both advertent and inadvertent loops. It also discusses some communication security, like denial-of-service attacks.

Section *5 Related work* gives references to other works on orchestration and choreography of cooperating Web Services. However, most published work is about static and rule based choreography.

Section *6 Conclusions* is an executive summary of the whole report. The last paragraph is quoted: *The bottom line is that our examinations and experimentations with the model have led us to confidently state that the described model is an adequate basis for the implementation of cooperating Web Services. The main merits are robustness and flexibility. It provides for tracing of the identities of all services involved, which builds up trust in the results, and it is particularly appropriate for tasks that can be characterized as "best effort" tasks.*

2 Part I: Model and background

2.1 Why Web Services?

Web Services has become the most prevalent solution for connecting systems in the most flexible way. Among the advantages of Web Services, compared to other technologies, is their independence of platforms and implementation languages. When it comes to connecting systems flexibly, a great advantage is that the connections are based upon message passing of relatively simple and self-contained messages in XML-formatted plain text. Among the drawbacks of Web Services is inefficiency in various respects, which is a price for flexibility. Another drawback is the uncertainty of the security for the resulting interconnected system of systems. Is it, for example, possible for a malicious system to take part among the interconnected systems without being noticed?

To preserve the important advantage of platform independence it is essential that the evolution of Web Services is governed by open standards. The two most important standardization bodies are OASIS [OASIS], for standards on the application level, and W3C [W3Ca], for standards on a more technical level.

For descriptions on which parts that make up a Web Service, we refer to [W3Ca] and, in Swedish, to [BEN03] and [BEN04]. We summarize what is needed in this report:

- Information shall be expressed as XML-elements. This applies both to messages sent between Web Services and to other documents, like description and declaration of elements. We assume that the reader is familiar with the basics of XML, for a short introduction see [W3Cd].
- The transport of messages between Web Services shall be carried out by a standard Internet protocol. In the rest of this report we assume HTTP or HTTPS but other protocols, like SMTP, are applicable.
- A message between Web Services shall be structured as one XML-message. There are some options, but we assume the most common standard, SOAP [SOAP].
- A Web Service is described in an XML document, following the standard WSDL [WSDL].

2.2 Security in Web Services

Security is potentially a stopper for the whole concept of open Web Services. In [IBMa] there is a roadmap for security within Web Services. Like in all information systems, the basic security functions are encryption and digital signatures. A digital signature of a message guarantees that the message is original and has not been manipulated. The signature is also used for authentication, that is verification of the identity of the sender of the message. As usual, the signature only verifies that a particular secret key was used when the signature was calculated. To verify an identity, a secure way to tie a key to an identity is also needed.

The standard for digital signatures of XML-formatted messages is XML-Signature Syntax and Processing [W3Cc]. This standard describes three ways to construct a digital signature. In this report the method called detached digital signature is used, since that is the way stipulated in [WSSe] for digital signing of elements in a SOAP-message.

Figure 1, collected from [W3Cc], outlines the XML code for a detached digital signature.

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

Figure 1: Detached digital signature.

The content of figure 1 is not a well-formed XML-message, but an outline. The question mark ‘?‘ denotes an element that can occur 0 or 1 time, the star ‘*‘ denotes an element that can occur 0 or many times and the plus ‘+‘ denotes an element that must occur 1 or many times. The signature itself is typically placed as an element in the SOAP-header. Within the element `<SignedInfo>` you find all the elements that are signed, among them refer-

ences to elements placed anywhere – in the SOAP-header, in the SOAP-body or anywhere that could be found by an URI, Universal Resource Identifier. This is why the signature is called detached.

2.3 Scenario

In order to have a tangible illustration of cooperating Web Services, we will use a military flavoured scenario throughout this report.

A network for Command & Control comprises several nodes. The larger part of the network is made up of clients, the rest are Web Services. In the scenario and figures 2-4 the Web Services are named A, B, C, …, U, V. The initiating client, X, has a request which, for instance, could be a question about intelligence observations in the area of Scania. This request is sent to A, a Web Service which authenticates X and issues an assertion that X is authorized to receive the information. A also knows other Web Services, named V and B, that can contribute to the task. V might be a unit for air-raid warnings and A invokes V with the subtask "X, with URL so and so, needs to know which aircrafts are heading towards Scania. I certify that X is authorized". B might be some surveillance resource which is invoked with "Which enemy objects are observed between coordinates so and so? Notify X, whose authorization I certify". B might know that C at the moment is better to handle this subtask and relays the request to C, perhaps including some data that C could use for its processing. C knows that D is in possession of relevant information and therefore invokes D. C also happens to have an Unmanned Aerial Vehicle (UAV) available for reconnaissance, but it will take some time before this UAV can return any observations to C. This can be modelled as C sending itself a one-way message, to invoke itself. This example scenario is depicted in figure 2. The dotted arrows are examples of partial results sent to the requesting client X.

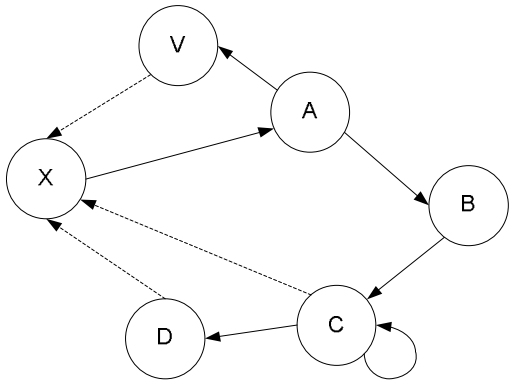


Figure 2: The scenario. A circuit of one-way messages.

2.4 Why a circuit?

The original model of how to connect to a Web Service was mainly a traditional client/server model. A client, most often controlled by a human via a GUI, issues a message to the Web Service in the form of a request for a service and then the client awaits the response. Subsequently, this model has evolved in different ways. The requested service can, like in our scenario, be in the form of a task that has to be jointly carried out by a set of cooperating Web Services. A lot of interest and research is focused on how to model such cooperation. Deciding which services that are to cooperate, and which service is doing what, is called orchestration [PEL03]. The way to communicate and how to send messages between the parties is called choreography [PEL03].

In [BEN05] we elaborate on three different choreographies applicable to our scenario – simple two-way messages, chained two-way messages, and a circuit of one-way messages.

The typical way for computers to communicate is in the client-server form. A client starts the communication by contacting the server with some kind of request. The server answers the client's request and by that fulfils the communication session. If the client has got several requests it may have to contact several servers in order to complete its needs. This situation is depicted in figure 3.

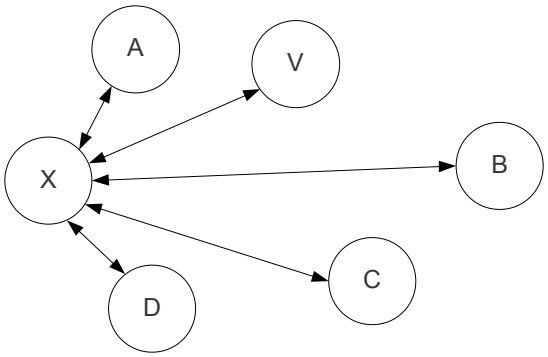


Figure 3: Simple two-way messages.

This choreography puts all the responsibility and coordination on the client (node X in figure 3). It is X who must decide whether or not to send a new request after receiving an answer. X must also know every useful Web Service in order to send additional requests. From a security point of view, X also has to be able to embed assertions of authority from previously invoked Web Services. All in all, X will have full control over the

transactions, but must also be equipped with enough functionality to handle all the communication. The invoked Web Services in this scheme are stateless since they can forget a request once they have answered it.

In the choreography depicted in figure 4, the client connects to a Web Service which will act as an agent for the client and which has an ability to fulfil a more complicated request. In some way, the situation is similar to client-server choreography with two-way messages. In this case though, a server may choose to become a client by forwarding the request to another Web Service, thus creating chained two-way messages.

When studying the figure, it becomes clear that each Web Service invoked in the request must remember the request and where it originated from. For a server with few requests this might not be a problem, but in a system with many services it is a major weakness. The robustness of large systems, for instance the Internet, is largely a consequence of keeping the communication as stateless as possible.

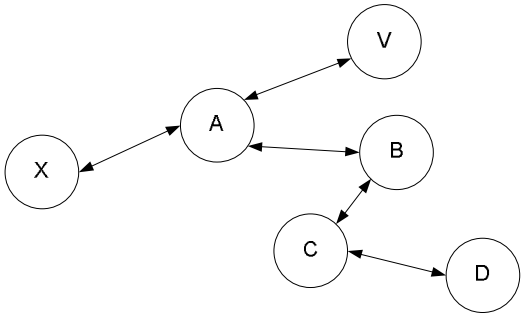


Figure 4: Chained two-way messages.

The choreography proposed by us is the one depicted in figure 2. It is based on asynchronous one-way messages. It forms a circuit, possibly with parallel loops. Each Web Service decides what to do next, and delivers one or many one-way messages. After a message has been sent the Web Service can forget about the request, thus becoming a stateless Web Service. It is argued here that the statelessness of the services is an advantage. X must, however, be capable of tracking the state of the task. A way for X to track, rather than remember, the state of a request is presented later in section 2.7. Compared to the traditional choreography, figure 3, X will not know when, or in what order, answers will arrive from other Web Services. Thus, X has less control, but is on the other hand not required to control everything. The proposed choreography, with one-way circuits, is particularly valuable in applications doing information searches. The result of an information search

is of “best effort”-nature, which means it is difficult to predetermine what will count as an all-inclusive search.

Note that the one-way messages preferably are sent by a standard two-way protocol like HTTP. Thereby the sending party gets a short receipt stating that the message has been delivered. Even better is HTTPS, which provides authentication of the communicating parties. Another comment is that also figure 2 can be modified, so that A is an agent for X. In that case the dotted arrows terminate at A instead of X. A will respond to X, either with a final response or with partial responses. Either way, this does not change the basics of the approach.

2.5 Why tracing?

To achieve trust in the information, it is, as in any communication, important to know the identity of the calling node. This is conventional authentication. In our proposed circuit of messages, it is equally important for X to know the identities of all the nodes which have participated in the circuit. For example, in the scenario X is never in contact with B, so X might not even know that B has been involved. If B were a malicious service, it might introduce false information into the execution and then hide itself. Another possibility is that C might want to make X believe that it was invoked directly by A. We therefore want a way for X to incrementally build a state, confirming the identities of the services invoked thus far.

The only opportunity for X to construct a state is when X receives results from services, at the dotted arrows in figure 2. Our approach is to create a data structure as a part of the one-way messages, sent when a service is invoked. Each service adds an element to the data structure, which thus is strictly growing for as long as the request is passed on. The data structure is of a form that makes it very natural for X to map it into a hierarchic tree. This tree tells X to what degree the task has been executed, and serves as the state of the task. The elements are digitally signed to facilitate authentication of all participating identities, thus creating trust in the information.

2.6 The data structure

To facilitate secure tracking of involved Web Services, each service adds elements to a strictly growing data structure and digitally signs the whole structure. The approach is to package all that is added by a service into a block. An example with three blocks can graphically be depicted as in figure 5.

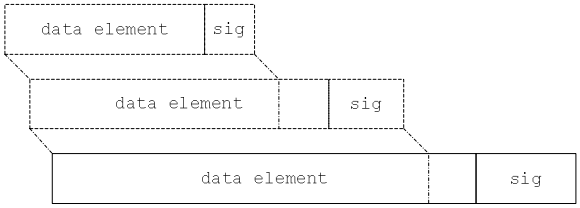


Figure 5: Blocks in data structure.

The signature `sig` means the digital signature of everything before. Web Service number k, let us call it WS_k, shall add the data that it creates, such as `<from>`, `<self>` and `<to>+` described later in section 2.7. WS_k shall also sign this data plus the data and signatures from k-1 earlier services. The essential XML-structure will be like figure 6.

```
<state>
    ...
    ...
    <block   ...  ID="uuid_b_{k-1}">
         .....
    </block>
    <block ...    ID="uuid_b_k">
        <ws ...  ID="uuid_ws_k">
            ...data from WS_k (from, self, to+)...
        </ws>
        <signature>
            .....
            <Reference URI="#uuid_b_0">
                ...
            </Reference>
            ......
            <Reference URI="#uuid_b_{k-1}">
                ...
            </Reference>
            <Reference URI="#uuid_ws_k">
                ...
            </Reference>
            .....
        </signature>
    </block>
</state>
```

Figure 6: Blocks in XML-structure.

The identifiers of `block`-elements and of `ws`-elements are proposed as uuid´s, Universally Unique Identifiers [LEA05]. They are referenced from the `<signature>`-element (from within `<SignedInfo>`). It should be

stressed that these uuid´s are identifiers of the XML-elements and should not be confused with identities of services.

The data structure leaving node A in the scenario, compare also figure 8, would essentially look like figure 7. The first block `<rootblock ... ID=”uuid_b0”> ... </rootblock>` is special. It is created by the requesting client, X. It is special in that X creates a block 0 regarding the task itself.

This XML-message maps the state of a communication. As such, it ought to be included in the header part of SOAP messages. On the other hand, it could be placed in the SOAP body, to be decoded only by services in a particular area, e.g. military intelligence.

It should be stressed, that the described data structure is only a part of the total SOAP message. The information sent and received from and to each Web Service and client makes up for most of the SOAP message. The structure described here supports secure tracing of identities and mapping of the state of the task into a hierarchic tree, as will be described in section 2.7. There will be additional elements in the total message, like inputs and results to and from Web Services. These elements might also be digitally signed, to achieve data authentication. These signatures, however, are not inside the structure in figure 7, since they are not relevant for the state of the task.

```
<state>
    <rootblock    ...  ID=”uuid_b0”>
        <task    ID=”uuid_tasknumber”>
          <client> X_id </client>
          <time> timestamp </time>
          .....
        </task>
    </rootblock>
    <block ...    ID=”uuid_b1”>
        <ws ...  ID=”uuid_ws1”>
            <from> null </from>
            <self> X_id </self>
            <to> A_id </to>
            etc ....
        </ws>
        <signature>
            .....
            <Reference     URI=”#uuid_b0”>
                ...
            </Reference>
            ......
            <Reference     URI=”#uuid_ws1”>
                ...
            </Reference>
            .....
        </signature>
    </block>
    <block ... ID=”uuid_b2”>
        <ws ...  ID=”uuid_ws2”>
            <from> X_id </from>
            <self> A_id </self>
            <to> V_id </to>
            <to> B_id </to>
            etc ....
        </ws>
        <signature>
            .....
            <Reference     URI=”#uuid_b0”>
                ...
            </Reference>
            <Reference     URI=”#uuid_b1”>
                ...
            </Reference>
            <Reference     URI=”#uuid_ws2”>
                ...
            </Reference>
            .....
        </signature>
    </block>
    .......
</state>
```

Figure 7: XML-structure leaving A.

2.7 The hierarchic tree

As mentioned earlier, X should not have to remember states of the request. X still has to be able to track a request though, in order to be able to verify the integrity of the responses. The data structure, of figures 5-6, makes it possible to securely track the identities of all Web Services involved in the execution of a task. One element in the structure is an identification of the task - a task number. This is used by the requesting client, X, to group partial responses together and to iteratively build a hierarchic tree.

The data structure is included in each one-way message sent. Each involved Web Service adds elements to the structure, which thus is strictly growing. Moreover, each Web Service digitally signs the whole structure. This means that X (and potentially also other Web Services involved) recursively can verify the identity of all parties that have been involved so far. Each service adds the identity of the Web Service that invoked it, of itself and of the Web Service(s) it, in turn, will call. Using the conventions from section 2.2, the elements can be denoted as `<from>`, `<self>` and `<to>+`.

The data structure, depicted as a hierarchic tree, at some stages in the example scenario, can be illustrated as figures 8-10.

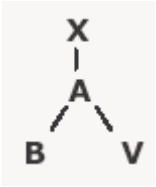


Figure 8: Tree built by A, sent to V and B.

Figure 8 provides the following information: Web Service A, which was the Web Service that X initially called, has in turn called Web Services B and V. None of these has yet answered the call. This means that this tree cannot be built by X, only by A, B and/or V.

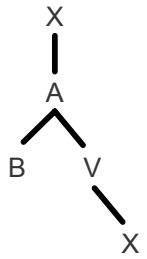


Figure 9: Tree sent from V to X.

When X has the information to build the tree in figure 9, X will know that Web Service V has answered the call, and that Web Service B also has been called. Thus, X has reason to believe that more answers will come. When X can put itself as a leaf in the tree, X will know that this particular branch is finished. As long as there is another name as a leaf in the tree, X can expect yet another reply of some sort.

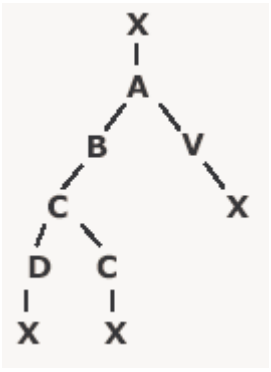


Figure 10: Tree of complete task.

Figure 10 shows a complete tree. At this stage X can see itself as a leaf in every branch, thus X is certain that no more results will arrive.

In addition to the identities `<from>`, `<self>` and `<to>+` , there will be other elements in the data structure. For instance, time stamps would serve two purposes. Firstly, they hamper replay attacks, when a message is sent repeatedly in some sort of denial-of-service attack. Secondly, X would need a time stamp to be able to decide how long to wait for more partial results to come. A great advantage of the approach presented here is that X can decide to act before the task is completed. This adds to the robustness of the overall system when a service temporarily is unavailable, when a message is delayed etc.

The calling service signs which other Web Services it calls (<to>+), as well as which Web Service invoked the calling service (<from>). In doing so the calling service closes the link between sender (<from>) and receiver (<to>+), thus preventing different types of man-in-the middle attacks. This is assuming that no secret key for signing has been compromised. A compromised key always allows masquerading. Furthermore, since the caller signs <to>, no Web Service can hide that it is involved. However, it is possible to hide loops in the chain. Any Web Service might for instance ask some Web Service Q for data that Q sends back to the service, without telling so.

2.8 Discussion

In 2.4 we discuss the merits of our proposed choreography, and two other choreographies used today. To recapitulate:

The traditional client/server, in figure 3, is often the most “natural” model. It is simple, which by itself is favourable for security, and it gives the client, X, full control. The negative side, from a security point of view, is that all clients must be able to authenticate and authorize themselves to all servers. The traditional model also means that the clients, X, must be very competent since they have to formulate all requests themselves.

The model, illustrated in figure 4, can be summarized as each service will act as an agent for the calling node (a client or other Web Service) to fulfil a task. The agent calls other services which, in turn, call yet other services and thus creating a chain. The partial results form a complete answer as they are assembled through each node in the chain. The initial node in the chain, A, then deliver the result to X, which means that X is relieved. The model is intuitive when a task consists of subtasks to be executed in a determinative way, as a kind of “all or nothing” task. It is implemented in e-business, for instance. Among its drawbacks, is that it introduces state interdependencies among the services.

Our proposed model in figure 2, a circuit of Web Services, is primarily natural in tasks that can be described as “best effort”, rather than “all or nothing”. It is a very flexible model. However, flexibility always has a price, in that controllability is reduced. We argue that our model provides two properties, essential for controllability and security. Firstly, the client is aided in determining the completeness of the task. Secondly, the identities of all contributing services are authenticated. Since the development of our model is in an early stage, there are more properties that should be scrutinized.

The proposed model, one-way messages in a circuit of Web Services, is in an early stage of research. It has been presented in conference papers [BEN05], and has been both favourably and unfavourably reviewed. One comment was that our model introduces a data structure, which every Web Service must be able to interpret and add element to. This is not realistic on a global level, world-wide. Since the model is in an early stage and not standardized, that is absolutely true, at least for a foreseeable future. However, this does not disqualify a system, like a nation's command & control system, to use Web Service technology to implement not yet standardized features. Our data structure can be placed in the body of the SOAP-message, thus it can be treated as ordinary information. Alternatively, it can be placed in the SOAP-header since a Web Service should ignore a header which it does not understand. However, our implementation (see part II) taught us, that high level platforms for implementation of Web Services do not permit manipulation of headers.

Another comment was related to X's lack of control. What if X received contradicting results from different services, or a result from a service that X did not trust? Although the problem is real, the ability to choose which service is more trustworthy is independent of the communication model. However, this is why it is important that our model gives X authenticated identities of the services, so that X will have the ability to decide the trustworthiness of the result.

A more relevant comment on X's lack of control is how the collaborating Web Services should know which service to call next, and how to call it. This is a hard problem, but it is also essentially independent of the communication model. Either X, or perhaps A, knows the rules for how to execute a task. These rules can be relayed as data between the services. If X or A does not know the rules, you are badly off in any model. The circuit model however provides the best result, since it is suited for a "best effort" task.

A correct comment on the circuit model is that the clients, X, also must act as servers, since they must listen to results arriving asynchronously. This means that the clients, in addition to the Web Services, are at risk of hostile calls, like denial-of-service attacks. These matters are further elaborated in part III.

In part III we also discuss another matter, namely looping. A special type of request to a service is subscriptions. In our scenario in section 2.3, a prevalent type of service might be a sensor, e.g. a radar sensor, which can send X updated information at regular, or irregular, times. Such a sub-

scription is naturally modelled as loops in the circuit model. In the agent model, illustrated in figure 4, this is harder to model. It is also unnatural in the traditional model, figure 3, since it requires X to poll the sensor. Looping has some consequences that must be dealt with. One is the impedingly long data structures, which would be a result of a long loop. Another is that there must be means for X to break a loop.

Finally, of course, we do not argue that all tasks should be executed as a circuit of services. In large system of systems, there will be different kinds of tasks, best executed by different models.

3 Part II: Implementing the trace

This section describes the experiences made and the conclusions drawn from the implementation of our model. The implementation was divided into two parts: a Web Services communication module (which included SOAP parsing and node tree-building), and a signing module (used to sign and verify signatures at the XML level of the messages). The clear distinction between the modules made it easy to develop them independently. The following sections describe the implementation of the model.

3.1 Requirements

3.1.1 Implementation goals

The implementation had the following goals:

- Implement tracing of Web Services with the help of signatures
- Multiple signatures should be added in a sequence in the SOAP header, following the syntax given in figure 7.
- The implementation should be able to defeat three different attack types to prove that the trace mechanism works. These are further described in section 3.1.2.
- Implement a demonstration that is easy to follow and understand. States and decisions in the node tree should be possible to control.
- Test the Web Services libraries and API's to see what they can provide in means of functionality when used in an alternative way, i.e. used in a way they were not designed for.

Some technical requests from previous work:

- Use SOAP
- Use WS-Security standard
- Use WASP Java server

The technical requests are a legacy from previous work, and are meant to be used if they still prove useful.

3.1.2 Attacks

One goal of the project was to demonstrate the trace mechanism during attacks. Three attack types against the trace signature mechanism were identified.

In the normal case each node appends its trace data together with a signature to the trace header, see figure 11.

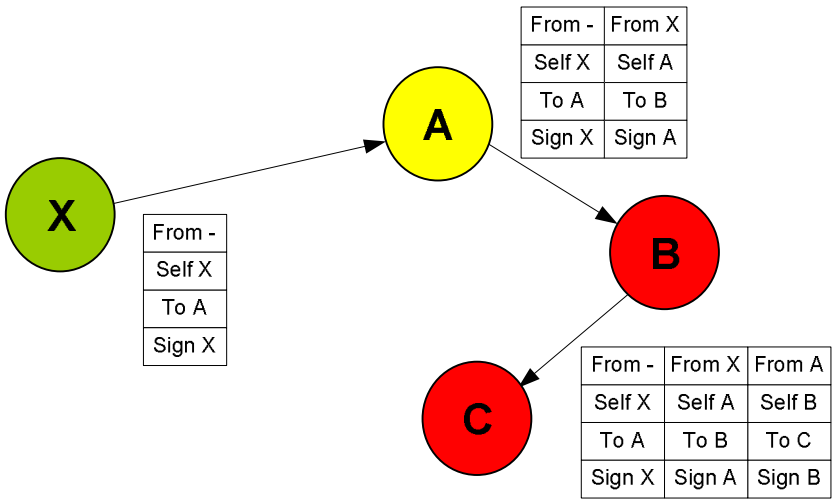


Figure 11: Trace headers in the normal case, with no attack.

The first type of attack, here referred to as integrity attack, tries to modify the contents of the trace header signed by another node. This will be detected when validating the signatures in the message with the public keys of the original senders.

The second type of attack, called man-in-the-middle, occurs when a node lies about its identity by changing the identity field denoting the sender (see the result from node B, in the last appended self field in figure 12). A better name for the attack could have been spoofing-attack, since the hostile node can assume any node identity in this way. The attack can be detected by comparing the given identity (of A), with the public key (of B), and with the registered certificate (of A), which will not match since the malicious node (B) does not have the key pair that correspond to the faked identity (A). If only the signature made by the public key supplied by B (without checking the identity) is naively verified, the signature will validate correctly and the attack will succeed.

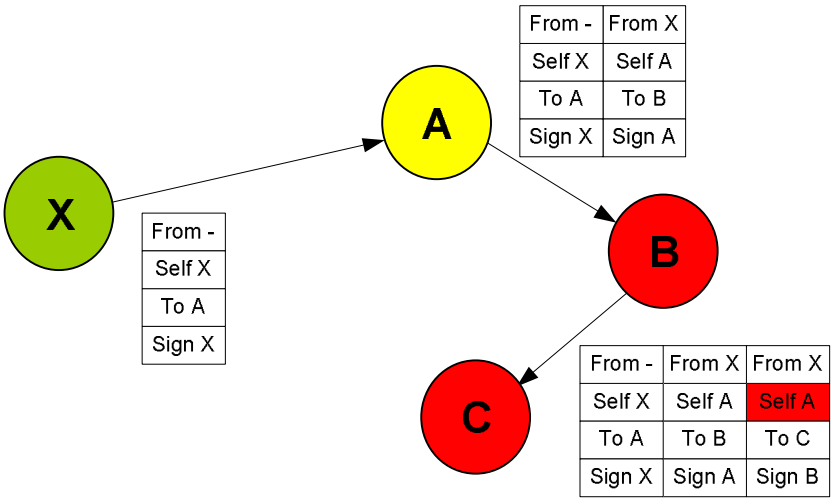


Figure 12: Man-in-the-middle attack.

A third type of attack was discussed during the project: removal of the previous node in the trace information chain, see figure 13. Simply deleting the information block from A will not do though. If done properly, the attacker B has to tamper with the original block sent from X to A, since B wants to be the direct receiver. B does this by replacing A with itself in the <to> field from X. This will be detected by the signature validation and is a special case of the integrity attack.

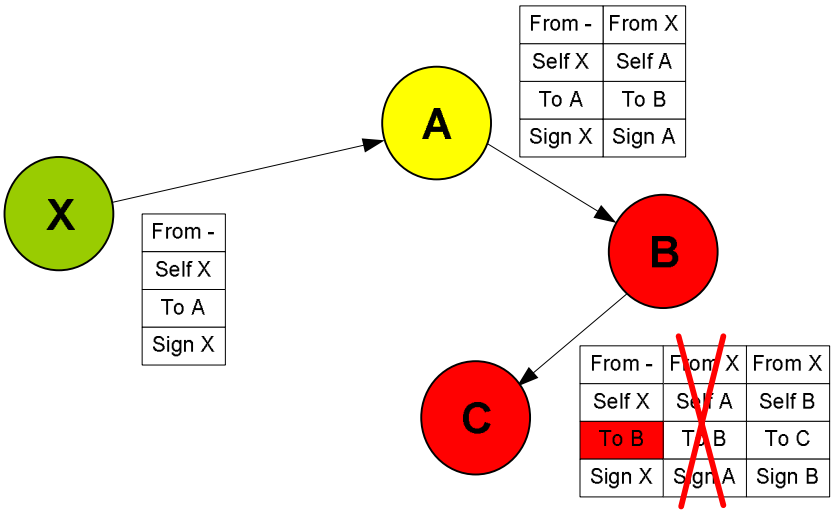


Figure 13: Removal of previous node information, and integrity check.

In a variation of the attack, B just removes A's trace information without tampering with X original trace information. This cannot be detected by

validating the signatures, but a hole will be found when traversing the receive-sent chain of all the nodes. In this case, X will detect the attack when inspecting the node tree in the packet that returns with the trace information back to X (see figure 2).

These three types of attacks are some examples of what the trace mechanism can achieve. There are of course more attacks possible like a node dropping the packet with no action, routing it back the wrong way, or claiming it is the last node and sending it to the originator X. Only the first three types of attacks, however, were considered in this report.

3.2 Design

3.2.1 Platform selection

The initial goal was to use Wasp Java server and the WS-Security standard. The motivation for our final selection follows.

Systinet Wasp server

Previous work done in the project used the Wasp-server as Java platform, see [HEI04]. Therefore a prototype for the trace mechanism, without the signature handling code, was created on the Wasp platform. The main idea was to add the signature code to the prototype, but this proved to be a way with many obstacles.

Wasp provided three different levels of interface functionality for the application programmer, with different abstraction levels:

- Raw Services (lowest level, provided most freedom in functionality, but had a more complex API)
- XML/SOAP Services
- Java Services (highest level, hid all advanced features, but was easier to use)

Since each level used its own classes and objects to model the data, it was not possible to mix levels, and the programmer was stuck with the classes that the data was generated or parsed with. A solution to the problem would have been to use the same level of functionality for all software (i.e. XML/SOAP). An obstacle, though, was that Systinet did not support the addition of multiple references, the ability to sign them simultaneously, and to place the signature in the header of the SOAP-message. The application programmer has no choice but to use the given API's for regular signing; sign a simple SOAP-message and place the signature after the message.

One way forward with Wasp could have been to install an alternative Web Services implementation package which allowed more programming freedom. Pilptchouk [PIL05] has made an example implementation of a WSSE (Web Services Security Extension) which provided an alternative API with more freedom. However, the problem to use another WS-Security implementation with Systinet was that the internal data structures had not been upgraded to DOM (Document Object Model [W3Ce]). Systinet had entangled themselves in the older format used in SAAJ v1.1 (Soap with Attachment API for Java [SAAJ]). A newer version of SAAJ, version 1.2, was available from Sun, but the latest Wasp server v5.5 still used the old version 1.1. Therefore no other recent implementations were available that would work with Wasp, which disqualified it as the platform.

Jakarta Tomcat Axis

The Jakarta Tomcat Axis server from the Apache project [AXIS] looked more promising since it consisted of parts from many different projects, and therefore was forced to use recent standards like DOM. Their WS-Security implementation was WSS4J v1.1 (Web Services Security for Java [WSS4J]). It supported multiple references, which should be possible to sign simultaneously. However, the control over where to insert the signature, as required by this project, was too limited. A disadvantage with Tomcat compared to Wasp was that smaller stand-alone services could not be deployed without starting the main server.

WS-Security

Both Systinet Wasp server and Jakarta Tomcat Axis have implemented WS-Security. It was, however, concluded that their implementation only provided limited functionality of the WS-security API, which was the most fundamental functionality that developers commonly used. Modifications needed to be made on a deeper level. The WS-Security did not even define a complete security solution; it was merely a foundation [DEI03]. In this project the services would send one-way messages forward in a chain which do not correspond to the normal client-server request-response model commonly used in WS-Security or in Web Services in general. There was also a problem extending the WS-Security standard with multiple concatenated signatures, there is usually just one signature in a message. The WSDL scheme of WS-Security did not allow a hierarchy of signatures. Thus, it became clear that WS-Security was not a suitable standard to go on with.

Sun JWSDP

Since WS-Security was not longer required, we went back to the roots and checked Sun’s XML security package which is a part of JWSDP (Java Web

Services Development Pack). It looked promising with many API's for both XML, signing, DOM and SOAP. This proved to be a good choice later on.

Webserver in Java

Instead of using the Systinet Wasp server or Jakarta Tomcat Axis server, we wrote our own web server in Java. It is a minimal server listening to incoming HTTP requests which it tries to parse into SOAP/XML messages using the parser in JWSDP. The server replies with a simple "202" message, which means it has accepted the incoming message.

The reason we built our own small server is that it is much easier to work with. The Wasp and Axis servers are pretty large and have lots of functionality which we had no use for. This made them slow to start up, and they used a high amount of system resources as each node would have to start its own server. Also, it was not easy to insert your own XML code into the messages in Wasp or Axis.

3.3 Implementation

This section deals with the programming phase of the project. It is assumed that the reader knows the basics behind signature creation and usage. If not, [BEN04] or [STA03] are recommended reading.

3.3.1 Java Modules

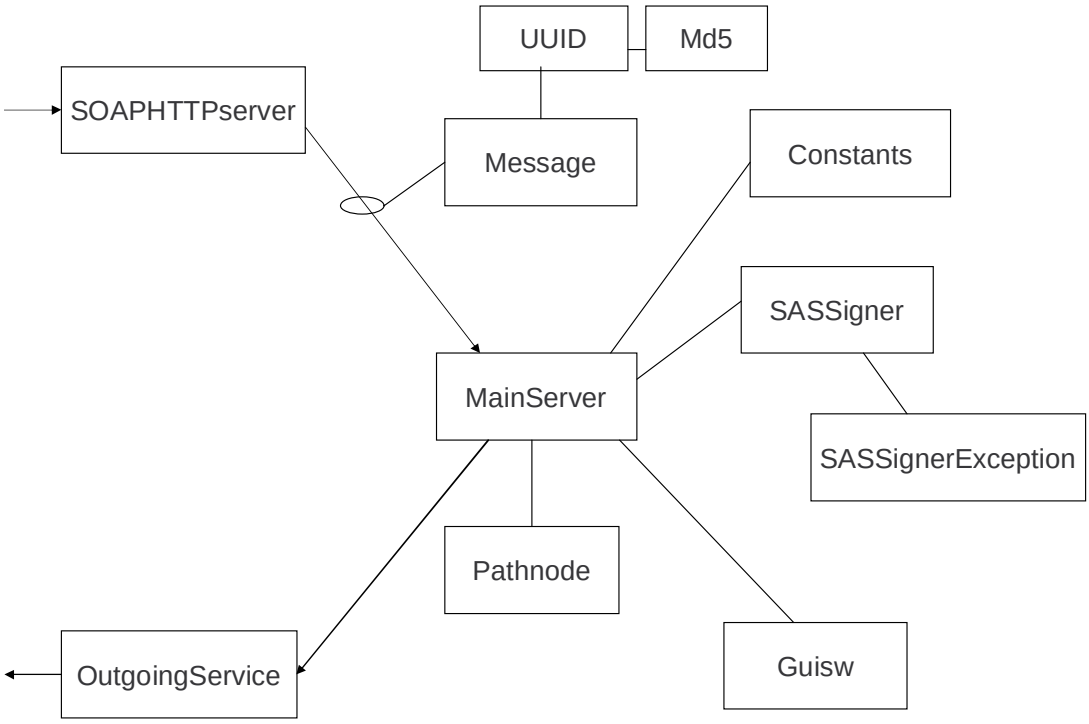


Figure 14: Java modules.

The implementation consists of several modules written in Java. The inter-relationships are shown in figure 14. The `SOAPHTTPserver` listens to the net and receives incoming SOAP messages. The messages are parsed into a DOM tree, put into the `Message` object and given a `UUID` which consists of several `Md5` checksums. The `MainServer` processes the message and sends it to `OutgoingService`. The `Constants` module stores the URLs, port numbers, and tag names for the blocks of XML code in the DOM-message. The `SASSigner` module signs or verifies the messages. `Guisw` is the graphical interface from where you can direct the sending and forwarding of messages. The `Pathnode` module is used when the chain is verified, and this is done by building a tree of the paths.

3.3.2 SASSigner

The signer module can either sign or verify signatures. To sign parts of a message, it requires a message from a file or as a tree representation in DOM format (`org.w3c.dom` in table 2) together with a list of references that

should be signed as well as the identity that should correspond with the private key. This might seem awkward, since a node (or user) has only one identity and one private key, but the solution provides a platform for testing the concept with various attacks.

The validation API only needs a message from file or in DOM representation; the identity can be retrieved from the XML message (`<self>`-field). If the identity matches the public key included (compared with the locally cached identity and public key, which we presume has been verified earlier), the key is trusted to use for validation.

The signature implementation identifies the first and second attacks described in section 3.1.2, and treats the third as an integrity attack when it can. When an attack is detected, the program internally communicates this with our own defined Java Exceptions, see table 1.

Table 1: SASSignerExceptions.

Attack exception	*Message description*
MIM	Wrong public key for an identity detected
INTEGRITY	Message has been tampered with

JWSDP packages used

The `SASSigner` module is implemented using JWSDP (Java Web Services Developers Pack, [JWSDP]) from Sun. The packages javax.xml.crypto.dsig and javax.xml.crypto.dsig.keyinfo provides a useful implementation of the W3C standard XML-Signature Syntax and Processing [W3Cc]. In table 2 there is a description of the packages and APIs used.

Table 2: Packages and API used from JWSDP.

Package	*Package description*	*API – description*
javax.xml.crypto.dsig	Sign and validate a digital signature	XMLSignatureFactory – Create signature object, add references etc. XMLSignature – Validate a signature SignatureMethod – Signature element Reference – Reference element SignedInfo – Signed info element

javax.xml.crypto.dsig .keyinfo	Parse and process KeyInfo element and structure	KeyInfoFactory – Create new KeyInfo objects KeyInfo – Contains XMLStructures to validate an XML signature KeyValue – Contains XML public key to validate signature
javax.xml.crypto.dsig .dom	DOM-specific classes for javax.xml.crypto.dsig package	DOMValidateContext – Specifies XMLSignature to unmarshal/ marshal for validation DOMSignContext – Species XMLSignature to unmarshal/ marshal for signature
javax.xml.parsers	Processing of XML documents	DocumentBuilderFactory – Parse DOM tree from an XML document
javax.xml.crypto	XML cryptography	KeySelector – Find key using data in KeyInfo KeySelectorResult – Return selected key selected by KeySelector XMLStructure – Groups XML structures, can be used to iterate over KeyInfo list
org.w3c.dom	DOM tree building blocks	NodeList – Ordered collection of nodes Document – Represents entire XML document
Javax.xml.soap	Provides the API for creating and building SOAP messages.	
Javax.xml.namespace	This package contains the QName class.	
Javax.xml.transform	This package defines the generic APIs for processing transformation instructions, and performing a transformation from source to result.	

3.3.3 Problems

The APIs given by JWSDP were adequate for the implementation, but did cause a problem when implementing the exception handling for the `SASSigner`. The idea was to generate a dedicated attack exception when finding one of the attacks. However, the man-in-the-middle attack was detected while processing an internal helper class based on the `KeySelector` interface. It was not possible to throw the exception from

within this class since it had already thrown another exception. Communication between the inner class (thrower) and the outer (catcher) had to be solved by using a global variable instead.

Problems with the ID attribute

During the testing phase of the implementation the following warning occurred when verifying a signature:

```
Aug 26, 2005 com.sun.org.apache.xml.security.utils.IdResolver
getElementById
INFO: Found an Element using an insecure Id/ID/id search method:
sas:rootblock
Aug 26, 2005 com.sun.org.apache.xml.security.utils.IdResolver
getElementById
INFO: Found an Element using an insecure Id/ID/id search method:
sas:ws
```

The problem is that when the signature verifier tries to find the ID attributes matching the references in the signature, it cannot actually be really sure that it found the right one. It is possible for an attacker to insert a bogus tag with an ID attribute and with the same reference UUID. This would result in the verifier finding the wrong DOM-node to verify. The warning above essentially says that it is not secure to only reference data elements by the value of an UUID-identifier. The whole document should also be validated with an XML schema, which would detect duplicate elements.

The attack as described, two elements with the same UUID, would be detected when the signature check fails. A more effective attack is to move the whole tag, and hide it inside some bogus element. Then you insert your own modified tag in its place, along with a new UUID. Consider the XML code in figures 15-16. Then the signature verification would be OK, since it refers to the original UUID-identifier. However, the chain built would be incorrect.

```
<state>
    ...
    <block   ...  ID="uuid_b_{k-1}">
        .....
    </block>
    <block ...    ID="uuid_b_k">
        <ws ...  ID="uuid_ws_k">
            ...data from WS_k (from, self, to+)...
        </ws>
        <signature>
            .....
            <Reference         URI="#uuid_b_0"> ...
            </Reference>
            ......
            <Reference         URI="#uuid_b_{k-1}"> ...
            </Reference>
            <Reference         URI="#uuid_ws_k"> ...
            </Reference>
            .....
```

Figure 15: Original XML code.

The XML-code in figure 15 can be modified according to the attack description (figure 16),

```
<state>
    ...
    <block   ...  ID="uuid_b_{k-1}">
        .....
    </block>
    <block ...    ID="uuid_b_k">
        <bogusblock>
            <ws ...  ID="uuid_ws_k">
                ...data from WS_k (from, self, to+)...
            </ws>
        </bogusblock>
        <ws ...  ID="new_bogus_id">
            ...modified data from WS_k (from, self, to+)...
        </ws>
        <signature>
            .....
            <Reference     URI="#uuid_b_0"> ...
            </Reference>
            ......
            <Reference     URI="#uuid_b_{k-1}"> ...
            </Reference>
            <Reference     URI="#uuid_ws_k"> ...
            </Reference>
            .....
```

Figure 16: Modified XML code.

The reference `uuid_ws`$_k$ now points to the real `<ws>` block inside `bogusblock` and the signature validation will verify it as correct. The chain will be wrong, since the newly inserted `<ws>` block with ID attribute "`new_bogus_id`" will have modified content. This will make it easier to perform the previously described attacks without being detected.

The solution to this problem is to validate the whole message with a XML schema before processing it. The validation will give an error since `<bogusblock>` is not an allowed tag name.

3.3.4 Creation of CA, certificates and keys

For the signature mechanism to work, all nodes need their own key pair; a private key and a public key. The key pair was created by a Certificate Authority (CA) which also packaged the public key in a certificate signed by the CA's own private key. This ensures validity of the public certificate.

The software package chosen to set up our own CA was OpenSSL [OPENSSL] which was available for a variety of platforms like Linux and WinXP. Some guidance for setup was given by [X509].

The default output from OpenSSL at the time was in PEM (Private Enhanced Mail, [BAL93]) format with the RSA (Rivest, Shamir & Adleman [STA03]) encryption algorithm, using a key length of 1024 bits. PEM is actually DER (Distinguished Encoding Rules [DER]) format encoded with base64 together with additional header and footer information. DER is a binary format, which most software in Java and WinXP recognizes. PEM, however, is easier to transfer, since it is in the ASCII format. An alternative to RSA could have been DSA (Digital Signature Algorithm).

Java (KeyFactory API) could not handle the default OpenSSL output format of the private keys; they had to be converted to PKCS8 (Public-Key Cryptography Standard number 8) in DER format. Number eight covers the private key syntax standard, see [PKCS]). This was done using the command:

```
openssl pkcs8 -topk8 -nocrypt -in x.key -out x_pkcs8_der.key
-outform der
```

3.4 Testing

To develop and test the `SASSigner` module separately, XML files following the specification in [BEN04] was coded by hand. A wrapper around the signer module was needed to act as the main program and for the ability to test it with different inputs. It parsed an XML message file into a DOM-tree,

built a list of references to be signed, and gave this information together with the identity to the signing module. By adding the number of nodes that traversed a message step by step, a set of legitimate and malicious XML messages could be created. Finally, the resulting files where used in a test suite, included in the wrapper.

The main program was tested by simply sending a text string between the nodes. Each node can print out debug messages showing the XML code for each message. The JDOM libraries were used to convert a DOM tree into printable text.

3.5 Debugging Web Services

It was easier to debug XML messages sent and received by the same machine (locally) when using Linux as compared to Windows. The reason for this is that Unix-based operating systems use a local loop back network interface, with IP-adress127.0.0.1. This port was monitored by the packet sniffer Ethereal [ETHERAL].

Microsoft has chosen another approach, and disabled this possibility. Instead, the developer is forced to use a proxy, which captures the traffic from the sender on one port and then retransmits it through the real (another) port to the receiver. Such a tool, SoapSpy, comes with the Systinet Wasp server v5.5 installation.

XML messages are best viewed with Firefox, which produces a simple and readable print of the message with automatically indented XML lines.

3.6 Demo

The main server has a graphical user interface, which is illustrated in figure 17. This interface is shown when one node (one server) is started. It consists of a node name in the title bar, and a text box where incoming and outgoing messages are shown. Below the text box is a smaller text field where the outgoing messages are inserted. A selection button (currently set to “A”) determines recipient and the “Send” button sends the message to the other node. The “Forward” line has a button and three drop down menus that is used to select which other nodes will receive the message when an incoming message is forwarded. One button verifies the signature and one button verifies the chain. A drop down menu is used to select which attack to introduce (blank in the picture). Finally, there is a “Quit” button.

Figure 17: Graphical user interface of X.

When the demo is started, several nodes are started at the same time with different names. It is possible to connect the nodes in any configuration by selecting different nodes on the “Forward” drop down menu.

3.7 Conclusion

The implementation resulted in the following conclusions.

- It is common in examples and tutorials for XML digital signatures to include the public key, and not the certificate, together with the signature. This is true in this implementation as well The X509Data class in JWSDP could have been used to add the certificate to the signature. At a first glance it might seem strange not to attach the certificate since it is easy to falsify a public key. If the certificate is not present, the public key cannot be validated locally towards the root CA certificate. On the other hand, if the public key is treated as an identity things start to make sense. There are two ways to handle public keys without certificates. The first is to cache trusted keys

locally (as in this implementation), and the second is to ask a trusted server to validate the key.

- The attacks described can be detected in different stages of the communication. The first, and most obvious detection, is that of a failed verification of a signature. However, by adding a bogus block, and thus hiding the incorrect UUID, the signature of the message may verify correctly. Thus, secondly, verification of a message by a XML Schema will be used to detect a modified structure of the message. Finally, verification of the complete chain will detect incompleteness in the path.
- The OpenSSL package worked well, but required a low-level command console to create key pairs and signing them. It provided format conversion commands to export it to a suitable format for Java. If OpenSSL is ever fully integrated to Java, it would open up new interesting possibilities for Java Web Services, and would certainly make things more user friendly.
- It was not critical how to solve key management in the implementation, since the goal was to test the Web Services API together with the trace mechanism. To cache all the certificates in a node is not scaleable, and would not work well in a real situation where certificates are revoked and new certificates are issued all the time. One solution to the scalability problem is to use PKI by using the XML Key Management Specification (XKMS). XKMS describes a Web Service for distribution, verification and registration of asymmetric keys. A basic implementation is described in [JOH05].
- It was easy to use the libraries of JWSDP in this implementation. We decided in the beginning of this project that we should base this security feature on WS-Security, but later discovered that WS-security was too restrictive about what type of extra XML code could be inserted into the WS messages. Instead, we only used the XML-DigSig package to sign the chain. This left us with a new question, where to put our chain of signed blocks - in the header or the body of the SOAP message? Either way is possible and, in most respects, correct. Adding information to the header may require more work that is outside the scope of this report and project, though. It was decided to put the signature chain in the body of the message.

The conclusion here is: when you want to be creative and make new security features with Web Services it is better to use building blocks like the Java libraries from Sun (or elsewhere) and not pre made platforms like Wasp or Axis. It is also difficult to follow restrictive standards when you want to introduce new features.

4 Part III: Additional security aspects

In part I we describe the primary elements of our model, which supports circuits of Web Services. Some basic parts have been implemented, as described in part II. In section 2.8 we mention some aspects that affect security, which is our main focus. Two aspects, looping and denial-of-service, are further elaborated here.

To improve readability, we repeat the figure of our scenario.

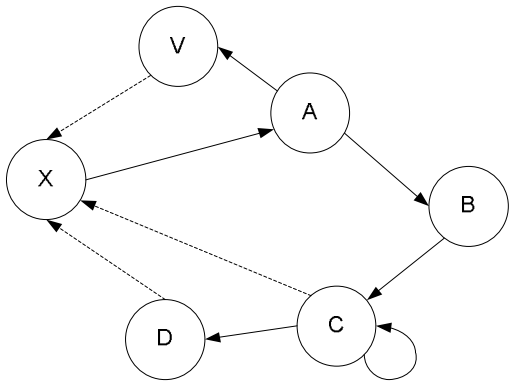


Figure 18: The scenario. A circuit of one-way messages .

4.1 Looping, data structure

Already in the main scenario, figure 18, there is a loop where node C calls itself. A situation, where this could be the case, is when C is a service which delivers updated data on a regular basis. The task, which X requests, might then include subscription of such updated data for some period of time. This is an example of a planned loop, where C knows that it will call itself.

The data structure in figure 19, handles loops in an intrinsic way. For each cycle in the loop, C adds a block like in figure 19 to the structure. The characteristic is that the same identity, `C_id`, is used in both `<from>`, `<self>` and `<to>`.

```
<state>
     ...
     ...
     <block   ...  ID="uuid_b_{k-1}">
          .....
     </block>
     <block ...    ID="uuid_b_k">
          <ws ...  ID="uuid_ws_k">
               <from> C_id </from>
               <self> C_id </self>
               <to> X_id </to>
               <to> C_id </to>
          </ws>
          <signature>
               .....
               <Reference     URI="#uuid_b_0"> ...
               </Reference>
               ......
               <Reference     URI="#uuid_b_{k-1}"> ...
               </Reference>
               <Reference     URI="#uuid_ws_k"> ...
               </Reference>
               .....
          </signature>
     </block>
</state>
```

Figure 19: Block in a loop.

The obvious negative consequence of adding a new block for each cycle is that the data structure may end up being impedingly long. Since XML is verbose, this results in unacceptable overhead. An extension to our model is to add an optional element `<loop_count>` in this situation, like in figure 20. C should then replace the last block from cycle number `n-1` with the block from cycle `n`, before signing the structure. It should be emphasized that X still can choose to build the hierarchic tree, with complete number of cycles. The overhead of a tall tree inside the application is less than the communication overhead.

```
<state>
     ...
     ...
     <block   ...  ID=”uuid_b_{k-1}”>
          .....
     </block>
     <block ...    ID=”uuid_b_{k+n}”>
          <ws ...  ID=”uuid_ws_{k+n}”>
               <from> C_id </from>
               <self> C_id </self>
               <to> X_id </to>
               <to> C_id </to>
               <loop_count> n </loop_count>
          </ws>
          <signature>
               .....
               <Reference     URI=”#uuid_b_0”> ...
               </Reference>
               ......
               <Reference     URI=”#uuid_b_{k-1}”> ...
               </Reference>
               <Reference     URI=”#uuid_ws_{k+n}”> ...
               </Reference>
               .....
          </signature>
     </block>
</state>
```

Figure 20: Added loop count.

In this example the loop is local, that is C is iteratively calling itself. Should it be that C in each cycle also calls another node, for instance D in figure 18 , the same principle can be used. Then, however, X is forced to build the tall tree, since D might act differently in each cycle.

Finally, the loop initiated by C might also involve one or several other nodes. For instance, the loop might be like C-Q-C-Q-C-... Then, C could add the element `<loop_count>`, and replace the blocks in the loop. Care must be taken, though, that the other node, Q in the example, has not acted differently in any cycle.

4.2 Looping, break

So far, only intentional loops, planned by C, have been discussed. Another aspect is detection of inadvertent loops. The decisions taken by Web Services how to involve other services can result in a Web Service being involved more than once in a task. This might be correct, but it might also

be an error that results in an endless loop. Some means to break an endless loop is needed.

One obvious possibility is that the Web Services build the hierarchic tree, and examine the identities. When an identity is found twice or more times, in a branch, a loop is detected. How to decide if the loop is planned or not, depends on the application. It is worth mentioning that this method does not mean that the Web Services must keep states. They can still be stateless, since all the needed data is included within each message.

Another possibility is that the client, X, breaks the loop. But this, more or less, violates the principle of one-way messages. In section 2.4 we remarked that the one-way messages preferably are sent using HTTP or HTTPS. The HTTP-layer gives an acknowledgement that the message was transmitted correctly. It is possible for X to use two different acknowledgements; “OK continue” and “OK break”, respectively [TAN03]. This could be detected by the sending service. However, some negative consequences are evident. It means interacting with the HTTP-layer, which is better avoided. It also means that a Web Service should wait for the acknowledgement from X, before other services, including itself, are called. This, in turn, is a security threat since it means that an obstructed acknowledgement could block a task. To sort this out, time-outs can be used, but that easily becomes complicated.

An alternative to HTTP-acknowledgements is to use real two-way messages. Then the acknowledgement would be in a separate SOAP message, essentially saying “Task `ID=”uuid_tasknumber”` continue/break”. In this case you do not interact with the HTTP-layer, but the concern about time-outs still remains.

All in all, there are alternatives to break loops. But they have to be scrutinized, in order not to introduce dependencies.

4.3 Communication Security

Two aspects of communication security will be discussed here. Firstly, the asynchronous nature of the circuit of one-way messages, and secondly, some aspects of denial-of-service.

One characteristic of our model, which we argue provides robustness, is that the one-way messages are sent asynchronously. This reduces dependencies which, in turn, improve robustness. Two classical security aspects of asynchrony are the effects of messages in the wrong order and of messages that disappear, respectively.

The model is insensitive to messages out of order. This is because the whole chain of preceding services is included in each message. It should be remarked, though, that there could be dependencies on sequence in the data element in the messages, but this is an entirely different matter.

For the same reason, the model itself is also insensitive to missing messages. It even provides some means for a client to figure out that a message is missing. If a service in the chain has declared `<to> X_id </to>`, and the client X has not received anything from that service, this message has disappeared. One problematic type of missed message, however, is missed acknowledgements, introduced to close a task or break a loop. This was discussed in section 4.2.

One comment on the model, circuit of one-way messages, is that also the clients, X, must listen and respond to incoming messages. This makes them susceptible to denial-of-service (DoS) attacks, in the same way as all Web Services are. Could that be mitigated?

The best way to mitigate DoS attacks is at the lowest possible level in the communication protocol stack. This would be at the TCP level. But this level is standardized and, thus, should not be modified by higher levels, like for instance our model. The only way to avoid denial-of-service is to stop listening to incoming TCP calls [TAN03]. This would mean that X cannot take care of asynchronous messages. A potential way, at least in some applications, to do this is to choose a proxy to buffer the messages. In our scenario, the service A acts as a main portal for X. It would be reasonable to expect that A could buffer the messages to X, which will poll A and ask if there are any messages buffered. The initial task request from X would then look something like figure 21.

```
<state>
     <rotblock     ...  ID=”uuid_b0”>
          <task    ID=”uuid_tasknumber”>
               <client> X_id </client>
               <time> timestamp </time>
               <proxy> A_id </proxy>
               .....
          </task>
     </rotblock>
     ....
```

Figure 21: Added proxy.

All the services in the circuit can see that X wants the responses to be buffered by A instead of sent to X directly. The rest of the data structure is unaffected.

The positive effect, the mitigation of denial-of-service, has to be balanced against the negative effects.

- It complicates matters for A, the proxy.
- It introduces time delays of the responses.
- The ways to close a task or break a loop, discussed in section 4.2, are obstructed.

If A should handle the loop control, the complexity would increase substantially.

Also higher communication levels, above TCP, can have some impact on denial-of-service. The messages in our model should be sent by a standardized internet protocol, like HTTP or HTTPS. Since the latter facilitates encryption and mutual authentication of the two communicating parties, it seems to be a good choice. But it has some drawbacks regarding denial-of-service. It is burdensome and, above all, the flow in the protocol [TAN03] is such that the first heavy computation is at the server side. This could make it easier for an attacker to choke the server. One suggestion, to change the asymmetry the other way around, is to use two messages instead. First, send a message via HTTP POST, essentially saying “I have a message to you concerning `<task ID=”uuid_tasknumber”>`, do you want it?”. Then the called service can issue a HTTPS GET to actually have the message delivered. Negative aspects of this idea are that complexity is increased and that two messages might introduce further security problems. A positive aspect is that the way to close a task or break a loop, see section 4.2, is made straightforward.

5 Related work

The approach in this paper has two main ingredients, choreography and security, respectively. Both of them are of great interest for the progress of Web Services.

Choreography means description of the modes of interaction between cooperating services. The most fundamental enhancement of the basic client/server model is asynchronous interactions. These permit the response to be delayed, as well as split into partial responses subsequently delivered. This is described in [ASYN]. (The basis is the same as proposed here; there is a task identifier to help gathering partial responses).

The concern of orchestrating Web Services to jointly execute a task is particularly obvious in e-business applications. In [Wf] work-flow is dealt with, how to describe separate parts of a process. The working group for choreography within W3C [WS-CDL] has published drafts for standards of languages and descriptions of choreography. They aim at descriptions from a global point of view, which is different from the approach here where the choreography grows in an ad hoc manner. Dynamic choreography, among peer-to-peers, is briefly mentioned in [WS-CDL].

An approach with evident similarities to this one is presented in [CAI04]. It is also aiming at decentralized and dynamic choreography. It is focused on how to describe the interactions in XML-formatted messages, which are passed along the cooperating services. Robustness and security aspects are not elaborated.

Security is potentially a stopper for the whole concept of open Web Services. In [IBMa] there is a roadmap for security within Web Services. On the basic level, regarding for instance digital signatures in SOAP messages, security is standardized [WSSe]. But it is not carefully elaborated on higher levels. At the choreography level, the emphasis is on reliable messaging, i.e. making sure that messages reach the recipient correctly. The approach in this paper is somewhat different, in that it focuses on “best effort”.

6 Conclusions

The purpose of this report has been to describe a model for tracing the identities in a set of cooperating Web Services. The Web Services are co-operating in a choreography that is characterized as a circuit of Web Services, communicating by asynchronous one way messages. This choreography is most natural in applications where a task is accomplished by the services executing subtasks in a dynamic way. The task is requested by a client, which receives the results from the subtasks in an asynchronous manner. The task can be regarded as a "best effort" task, as opposed to a more static rule based "all or nothing" task. An important example of "best effort" tasks is information searches. A military flavoured information search is used throughout the report as an example scenario.

Our motive for studying Web Services is that this is a candidate for the Service Oriented Architecture that is decided on for the next generation of Sweden's Command & Control System. Our focus is on security aspects, which are a potential stopper for the whole concept of Web Services in military systems.

We discuss the security aspects from two points of view. Firstly, to get trust in the information, the receiver of the results must be able to trace the identities of all participating Web Services. It should not be possible for malicious Web Services to hide or masquerade themselves. We therefore propose a data structure as part of the one-way messages. This data is digitally signed by each Web Service. Each service adds an element to the structure, which thus is strictly growing. Each added element ties sender and receiver together, which prevents masquerading, man-in-the-middle, etc. It also prevents a service to conceal its participation. An exception is that in some circumstances a service can hide a local loop. The signatures act as authentication of the identities.

The second security aspect is robustness, i.e. ability to withstand both inadvertent disturbances, like delays, and advertent disturbances, like denial-of-service attacks. We argue that our model has many advantages when it comes to robustness. The data structure, which is tied to each message, essentially means that the current state of the corresponding branch of the task is available in each message. This makes it possible to handle delayed and lost messages. It also means that the participating Web Services do not have to remember the state of the task. The services can be stateless, a major advantage to robustness. Furthermore, it also means that the client that requested the task can deduce to what extent the task has been

executed, thereby enabling the client an opportunity to act in a “best effort” way.

The benefits of the model must be balanced against some disadvantages, one being the introduced overhead. Loops in the circuit of Web Services can result in impedingly long data structures. Some ways to mitigate this are discussed in the report. Care must be taken, though, not to destroy the main benefits of the model, the robustness and the flexibility.

Our model is new and has not been implemented in a full-fledged application. This means that it is vital to implement the main parts of the model, to be able to reason about the usefulness of the model. The conclusion from our implementation is that the model itself is readily implemented, since it is based on standard Web Services properties, like XML-messaging and XML Digital Signatures. However, when you want to implement new security features, it is better to use building blocks like the Java libraries from Sun (or elsewhere) rather than pre fabricated platforms like Wasp or Axis. It is difficult to follow restrictive standards when you want to introduce new features.

The bottom line is that our examinations and experimentations with the model have led us to confidently state that the described model is an adequate basis for the implementation of cooperating Web Services. The main merits are robustness and flexibility. It provides for tracing of the identities of all services involved, which builds up trust in the results, and it is particularly appropriate for tasks that can be characterized as “best effort” tasks.

References

Literature references

[BEN03] Bengtsson A., Hunstad A. & Westerdahl L.: *Identitetsverifiering över systemgränser*, Användarrapport, FOI-R--1025--SE, November 2003

[BEN04] Bengtsson, A.: *Spårning vid samverkande Web Services*, FOI-R--1399--SE, November 2004

[BEN05] Bengtsson, A. & Westerdahl, L.: "Secure Choreography of Cooperating Web Services", in *Proceedings of the 3rd IEEE European Conference on Web Services (ECOWS 2005)*, Växjö, Sweden, November 14-16, 2005

[CAI04] Caituiro-Monge, H. & Rodríguez-Martinez, M.: Net Traveler: "A Framework for Autonomic Web Services Collaboration, Orchestration and Choreography in E-Government Information Systems", in *Proceedings of the IEEE International Conference on Web Services (ICWS´04)*, San Diego, California, USA. July 6-9, 2004

[DEI03] Deitel, H.M., Deitel, P.J., DuWaldt, B. & Trees, L.K.: *Web Services – A Technical Introduction*, Prentice Hall, 2003

[HEI04] Heinonen, M. & Manis Sörensen, C.:*Connecting Systems with Secure and Interoperable Web Services*", Linköping Institute of Technology, LiTH-ISY-EX3475-2004, 2004

[JOH05] Johansson, A.: *Utveckling av Web Service för hantering av öppna autentiseringsnycklar*, Linköping Institute of Technology, LiTH-ISY-EX- - 05 / 3747 - - SE, 2005

[PEL03] Peltz C.: "Web Services Orchestration and Choreography", *IEEE Computer*, Vol. 36, No. 10, 2003, pp. 46-52

[STA03] Stallings, W.: *Network Security Essentials (2nd edition)*, Prentice Hall, 2003.

[TAN03] Tanenbaum A.: *Computer Networks (4th edition)*, Prentice Hall, 2003

Web references

[ASYN] Asynchronous Transactions and Web Services, 10 November, 2003 http://xml.coverpages.org/async.html (visited 3 October, 2005)

[AXIS] Web Services – Axis, http://ws.apache.org/axis/ (visited 3 October, 2005)

[BAL93] Balenson, D.: "Privacy Enhancement for Internet Electronic Mail: Part III: Algorithms, Modes, and Identifiers", RFC 1423, February, 1993

http://www.faqs.org/rfcs/rfc1423.html (visited 3 October, 2005)

[DER] ASN.1 Standards, “ITU-T X.690” describes DER (Distinguished Encoding Rules), http://asn1.elibel.tm.fr/en/standards/ (visited 3 October, 2005)

[ETHERAL] http://www.ethereal.com/ (visited 3 October, 2005)

[IBMa] “Security in a Web Services World: A Proposed Architecture and Roadmap”, 7 April, 2002 ftp://www6.software.ibm.com/software/developer/library/ws-secmap.pdf (visited 11 November, 2005)

[IBMb] “WS-Security Profile for XML-based Tokens”, 28 August, 2002 ftp://www6.software.ibm.com/software/developer/library/ws-sectoken.pdf (visited 3 October, 2005)

[JWSDP] Java Web Services Developer Pack (Version 1.5) Combined API Specification, JWSDP java doc http://java.sun.com/webservices/docs/1.5/api/index.html (visited 3 October, 2005)

[LEA05] Leach P., Mealling M. & Salz R.: “A UUID URN Namespace”, RFC 4122, Internet Official Protocol Standards, July, 2005 ftp://ftp.rfc-editor.org/in-notes/rfc4122.txt (visited 3 October, 2005)

[LIB03] “Liberty Alliance & WS-Federation: A Comparative Overview”, Liberty Alliance Project white paper, 14 October, 2003 http://projectliberty.org/resources/whitepapers/wsfed-liberty-overview-10-13-03.pdf (visited 3 October, 2005)

[OASIS] Organization for the Advancement of Structured Information Standards http://www.oasis-open.org/who/ (visited 3 October, 2005)

[OPENSSL] OpenSSL Project http://www.openssl.org, (visited 3 October, 2005)

[PIL05] Pilptchouk, D.: “WS-Security in the Enterprise, Part1: Problem Introduction”, 2 September, 2005 http://www.onjava.com/pub/a/onjava/2005/02/09/wssecurity.html (visited 3 October, 2005)

[PKCS] “PKCS #8: Private-Key Information Syntax Standard”, *RSA Laboratories*, 1 November, 1993 http://www.rsasecurity.com/rsalabs/node.asp?id=2130 (visited 3 October, 2005)

[SOAP] “SOAP Version 1.2 Part 0: Primer” http://www.w3.org/TR/soap12-part0 (visited 3 October, 2005)

[SAAJ] SOAP with Attachments API for Java (SAAJ), http://java.sun.com/xml/saaj/index.jsp, (visited 3 October, 2005)

[W3Ca] Web Services Activity http://www.w3.org/2002/ws/ (visited 3 October, 2005)

[W3Cb] "Web Services Architecture Requirements", W3C Working Draft 19 August, 2002 http://www.w3.org/TR/2002/WD-wsa-reqs-20020819 (visited 3 October, 2005)

[W3Cc] "XML-Signature Syntax and Processing" http://www.w3.org/TR/xmldsig-core/ (visited 3 October, 2005)

[W3Cd] XML in 10 points http://www.w3.org/XML/1999/XML-in-10-points.html (visited 3 October, 2005)

[W3Ce] Document Object Model (DOM) http://www.w3.org/DOM/ (visited 3 October, 2005)

[WASP] Systinet Server for Java http://www.systinet.com/products/wasp_jserver/overview (visited 3 October, 2005)

[Wf] "XML-Based Workflow and Process Management Standards: XPDL, Wf-XML", 21 June, 2004 http://xml.coverpages.org/wf-xml.html (visited 3 October, 2005)

[WS-CDL] "Web Services Choreography Description Language Version 1.0", W3C Working Draft 12 October, 2004 http://www.w3.org/TR/ws-cdl-10/ (visited 3 October, 2005)

[WSDL] "Web Services Description Language (WSDL) Version 1.2 Part 1: Core Language", W3C Working Draft, 11 June, 2003 http://www.w3.org/TR/2003/WD-wsdl12-20030611 (visited 3 October, 2005)

[WSSe] "Web Services Security: SOAP Message Security 1.0 (WS-Security 2004)", 15 March, 2004 http://xml.coverpages.org/WSS-SOAP-MessageSecurityV10-20040315.pdf (visited 3 October, 2005)

[WSS4J] Apache WSS4J API Overview http://ws.apache.org/wss4j/api.html (visited 3 October, 2005)

[X509] "X509 CA Certificate generation" http://www.cornelius.demon.co.uk/X509-Cert-Generation.html (visited 3 October, 2005)

[XKMS] "XML Key Management Specification (XKMS 2.0) Version 2.0", W3C Candidate Recommendation, 5 April, 2004 http://www.w3.org/TR/xkms2/ (visited 3 October, 2005)