SOAP-based Secure Conversation and Collaboration

Mohammad Ashiqur Rahaman, Andreas Schaad
SAP Research
805, Avenue du Docteur Maurice Donat
Font de l'Orme, 06250 MOUGINS, France
+33(0)4 92 28 62 00
{mohammad.ashiqur.rahaman, andreas.schaad}@sap.com

Abstract

Web services in different trust boundaries interact with each other via SOAP messages to realize functionality in a collaborative environment. Exchanging SOAP messages for remote service invocation has gained wide acceptance among web service developers. Several web service security standards are widely deployed aiming at securing exchanges of a single SOAP message and a conversation of SOAP messages among partners in a collaborative environment. Concerns have been raised about the possibility of XML rewriting attacks within this context and their early detection.

In this paper, we demonstrate such possible attacks with respect to WS policy based scenarios to set a security context and to use a security context for conversations of SOAP messages. We show how our proposed `SOAP Account` [21] solution could be applied for early detection of XML rewriting attacks, specifically regarding secure SOAP-based conversations. A simulation-based performance analysis and comparison of our `SOAP Account` approach vs. a WS* policy based approach complements our observations.*

1. Introduction

Interaction among partners in a collaboration may be done using two classic techniques of a distributed system: shared memory and message passing. In service-oriented architectures (SOA) the latter approach has been used as a de facto standard for interaction among services.

Secure message exchanges among the partners in a collaborative environment may be required to find appropriate partners, set common goals and achieve a defined set of results. In a SOA, individual web services may reside in different organizational boundaries. Often, they do not have prior trust among them. We need a trust framework which will essentially bridge the gap of trust among partners in different organizations. Interaction among services, whether of a static or a dynamic nature, occurs through SOAP [7] message exchanges.

A series of web service standards: WS-Security [20], WS-Policy [22], WS-SecurityPolicy [6], WS-Trust [18] and WS-SecureConversation [17] address the mentioned concerns of *securing message exchanges* both in single messages and conversation of messages scenarios and establishing a *trust framework* across different trust boundaries. However, implementing any WS* standard in the underlying SOAP messaging layer, introduces new XML elements and thus enlarges the SOAP message size which in turn introduces overhead in terms of size and complexity of XML processing and/or cryptographic operations. A *performance analysis* with precise performance criteria would provide a measurable rationale regarding the deployment of WS* standards.

Our previous work [21] discusses the theory of a `SOAP Account` and the usage, limitations of policy to detect XML rewriting attacks and [16] addresses the same attack on a `SOAP Account` itself. However, in this paper[1] we focus on a *secure conversation framework* for which an underlying *trust framework* is complementary and how these frameworks are used in the current web service security architecture (e.g. SOA). Accordingly, we see the key contributions of this paper in:

- an overall understanding of applying WS-Trust and WS-SecureConversation in a collaborative environment;
- early detection of XML rewriting attacks with respect to different trust domains and conversations of messages;
- a detailed performance analysis of various performance issues including the detection time of XML rewriting attacks using a policy based approach vs. our proposed `SOAP Account` technique.

The rest of this paper is organized as follows. Section 2 describes trust and secure conversation frameworks as realized by different web service standards. We briefly refer to related standards (e.g. WS-Security, WS-Policy) where it is necessary with respect to our discussions. Section 3 demonstrates possible attacks (e.g. XML rewriting attacks) by means of two example scenarios. Section 4 shows how a `SOAP Account` can be useful to detect these attacks when

[1]The work of M. A. Rahaman and A. Schaad has been sponsored under the EU IST-2004-026650 project "R4eGov".

used in trust and secure conversation frameworks and how it may fit into current web service security architectures. Section 5 provides a performance analysis regarding a `SOAP Account` vs. a WS* policy-based approach. Section 6 briefly discusses related work and section 7 concludes the paper.

2. Trust and Secure Conversation

Trust is the characteristic that one partner (consumer, service) is willing to rely on a second partner (consumer, service) to execute a set of tasks and/or to make a set of statements/assertions about a set of subjects and/or scopes [18]. A *Claim* is any statement/assertion about a partner (consumer, provider of a service). A *Security Token* is a token which represents a collection of claims. A *Trust Engine* is a conceptual component that evaluates the security-related aspects of a message in a trust framework. A *Security Token Service (STS)* is a service, responsible for issuing/renewing/validating security tokens. *Request security Token (RST)* is a token in a SOAP message, sent to a *STS* to request a security token. *Request security Token Response (RSTR)* is a response token to the requestor containing the requested token.

The analysis of the WS* standards related to the trust and secure conversation frameworks is based on [18] and [17]. Trust and conversation is not a new concept in security. Numerous classic protocols, aiming at the mutual authentication of the partners involved in a conversation, exist in the literature and industrial deployments. Most of the concepts can be applied to WS-Security, WS-Trust, WS-SecureConversation, but experience shows that this adaptation is not so straightforward.

2.1. WS-Trust

WS-Trust [18] defines the trust framework that we need in a collaborative environment. While this framework is based on the mechanisms described in WS-Security [20], it defines additional primitives and extensions for security token exchange to enable partners to interact with SOAP messages within various trust domains. So, essentially trust is represented through the exchange and brokering of security tokens among partners. The trust framework provides:

- Methods for issuing, renewing, and validating security tokens;
- Ways to establish, assess the presence of, and broker trust relationships.

The overall process is as follows - one service may require that any incoming message prove a set of claims (e.g. name, key, permission, capability, etc.). Any message arriving without the required proof of claims should be ignored or rejected by the service. A service's required claims can be described using WS-Policy [22] and WS-SecurityPolicy [6] specifications. So, if any requestor of a service does not have the necessary tokens to prove the required claims, it may request to the appropriate third entity (according to the service's policy) for the required tokens with appropriate

```
<RequestSecurityToken Context="...">
  <TokenType>...</TokenType>
  <RequestType>...</RequestType>
  <Base>...</Base>
  <Supporting>...</Supporting> ...
  <wsp:AppliesTo>...</wsp:AppliesTo>
  <Claims Dialect="...">...</Claims>
  <Entropy>
    <BinarySecret>...<BinarySecret></Entropy>
  <Lifetime>
    <wsu:Created>...</wsu:Created>
    <wsu:Expires>...</wsu:Expires></Lifetime>
</RequestSecurityToken>
```

Figure 1. Requesting a security token `(RST)`

```
<RequestSecurityTokenResponse Context="...">
  <TokenType>...</TokenType>
  <RequestedSecurityToken>...
  </RequestedSecurityToken> ...
  <wsp:AppliesTo>...</wsp:AppliesTo>
  <RequestedTokenReference>...</RequestedTokenReference>
  <RequestedProofToken>...</RequestedProofToken>
  <Entropy>
    <BinarySecret>...<BinarySecret></Entropy>
  <Lifetime>...</Lifetime>
</RequestSecurityTokenResponse>
```

Figure 2. Response a security token `(RSTR)`

```
<wsc:SecurityContextToken wsu:Id="...".>
  <wsc:Identifier>...</wsc:Identifier>
  <wsc:Instance>...</wsc:Instance> ...
</wsc:SecurityContextToken>
```

Figure 3. Structure of Security Context Token

claims. The third entity is essentially a *security token service (STS)*, which may in turn require its own set of claims for authenticating and authorizing the request for security tokens. The *STS* forms the basis of trust by issuing security tokens that can be used to broker trust relationship among different trust domains. Service may have its own policy, receives a message from a requestor. Requestor may include signature, security tokens, using WS-Security [20] features. The *Trust engine* residing in the service performs a set of key steps before processing the request:

- Verify that the attached claims in the token are sufficient to comply with the policy and that the message conforms to the policy.
- Verify that the subjects and other security attributes of the claims are valid by the signatures.
- Verify that the issuers of the security tokens are trusted.

The *trust framework* is realized by a number of XML elements that are used to request security tokens *(RST)* and respond to the request security tokens *(RSTR)*. Figure 1 and Figure 2 show a typical structure of a token request (`<RequestSecurityToken>`) and response (`<RequestSecurityTokenResponse>`) elements identifying the general mechanisms respectively. Eventually these elements will be the payloads (in some cases these can be in the SOAP header) in the exchanged SOAP messages.

2.2. WS-SecureConversation

WS-SecureConversation [17] aims at the security of a conversation of related messages in a collaboration. While the WS-Security [20] focuses on the message authentication model, the WS-SecureConversation focuses on the context authentication model (security context model) for a conversation of SOAP messages. While the former model is used for securing a single message, the latter is useful for the partners wishing to exchange multiple messages. The security context is defined as a new security token type in

WS-Security that is obtained using the process described in the previous section about WS-Trust. Figure 3 shows the structure of a security context token. The goals of the WS-SecureConversation specification are:

- Establishing security contexts for a conversation of messages.
- Amending, Renewing, and cancelling the security contexts.
- Computing and passing derived keys and session keys.

WS-SecureConversation [17] left some terms and their relationships undefined. It does not specify any definition of a conversation although the title of the standard uses the word "conversation". It is also not clarified that how the context and conversation starts/ends and depends on each other. [10] also mentions these disparities of the WS-SecureConversation standard and makes its own clarification. In our case we use the terms "conversation" and "session" to refer to the same concept while the notion of starting/ending does not require further definition in our context.

A security context token must be created and shared among the partners before using it in the conversation of messages. There are three ways to create a security context token. A *STS* can create a new security context token after receiving a request. Alternatively, one partner can create a new security context token and shares it with the other partners. However, this way works only when sender is always trusted to create a new security context token. The previous two ways allow for a simple request/response for security context tokens. However, there are some scenarios where a sequence of message exchanges between partners is required prior to the issuing a security context token. One scenario could be that the partners may want to negotiate about the shared secret of the security context tokens which may be done with several message exchanges. Once the security context token is established it can be distributed to the partners for the lifetime of a communication session through the mechanisms described in WS-Trust. Amending and canceling a security context can be done using the same ways for establishing of a security context token. For establishing, amending and canceling a security context token a `RST` token for a request and a `RSTR` token for a response are used in a SOAP message.

An established security context token contains or implies a shared secret which may be used for signing and/or encrypting messages. However, partners may derive different keys using that shared secret key using a specific function [17]. For example, four keys may be derived so that two partners can sign and encrypt using separate keys. Using WS-Security features, like, attaching security tokens, token references, and secrets for each message in a conversation consumes considerable processing time for the SOAP processing nodes (e.g. requestor, partners). Establishing security context among partners and thereby having shared secret based on the context allows one to derive more keys. One can derive its own different required keys offline, and

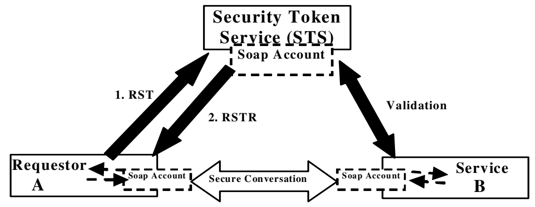


Figure 4. Secure Conversation of multiple messages without and with SOAP Account (Dashed box and Arrow)

```
<Envelope>  ← Message to get a security context token
 <Header>..
  <Action> http://schemas.xmlsoap.org/
    ws/2004/04/security/trust/RST/SCT
  </Action> ...
  <Security>
   <BinarySecurityToken Id=" Id-2" ValueType="...X509v3">
   MIIEZzCCA9CgAwIBAgIQEmtJZc0...</BinarySecurityToken>
    <Signature>
     <SignedInfo>
       <CanonicalizationMethod Algorithm="..xml-exc-c14n#"/>
       <SignatureMethod Algorithm="...#rsa-sha1"/>
       <Reference URI="#Id-1">
        <DigestMethod Algorithm="...#sha1"/>
        <DigestValue>d5AOd..=</DigestValue>
       </Reference>
       <Reference URI="#Id-2>...</Reference>
     </SignedInfo>
     <SignatureValue>e4EyW...=</SignatureValue>
     <KeyInfo>
     <SecurityTokenReference><Reference URI="#Id-2"
      ValueType="...#X509v3" /></KeyInfo>
    </Signature>
   </Security>...
 </Header>
 <Body Id="Id-1">                         RST token is signed
  <RequestSecurityToken>
    <TokenType>http://schemas.xmlsoap.org/ws/2005/02/sc/sct
</TokenType>
    <RequestType>http://schemas.xmlsoap.org/
     ws/2004/04/security/trust/Issue
    </RequestType>
    <Base>...</Base>
  </RequestSecurityToken></Body></Envelope>
```

Figure 5. Soap request with RST token before XML rewriting attack (excerpt)

```
<Envelope>
 <Header>..  ← Attacker has intercepted the message
  <Security>
   <BinarySecurityToken Id=" Id-2" ValueType="...X509v3">
 MIIEZzCCA9CgAwIBAgIQEmtJZc0...</BinarySecurityToken>
    <Signature>
     <SignedInfo>...
      <Reference URI="#Id-1">...</Reference>
      <Reference URI="#Id-2>...</Reference>
     </SignedInfo>
     <SignatureValue>e4EyW...=</SignatureValue>
     <KeyInfo><SecurityTokenReference><Reference URI="#Id-2"
       ValueType="...#X509v3" /></KeyInfo>
     </Signature>
  </Security>
  <BogusHeader>
   <Body Id="Id-1">  ← XML Rewriting attack
    <RequestSecurityToken>
     <TokenType> http://schemas.xmlsoap.org/ws/2005/02/sc/sct
</TokenType>
     <RequestType>
http://schemas.xmlsoap.org/ws/2004/04/security/trust/Issue
     </RequestType>
     <Base>...</Base>
    </RequestSecurityToken>
   </Body>
  </BogusHeader>
 </Header>                 Request for a custom token by the Attacker
 <Body>
  <RequestSecurityToken>
   <TokenType>http://example.org/myBogusToken</TokenType>
   <RequestType> http://schemas.xmlsoap.org/
    ws/2004/04/security/trust/Renew
   </RequestType>
  </RequestSecurityToken></Body></Envelope>
```

Figure 6. Soap request with RST token after XML rewriting attack (excerpt)

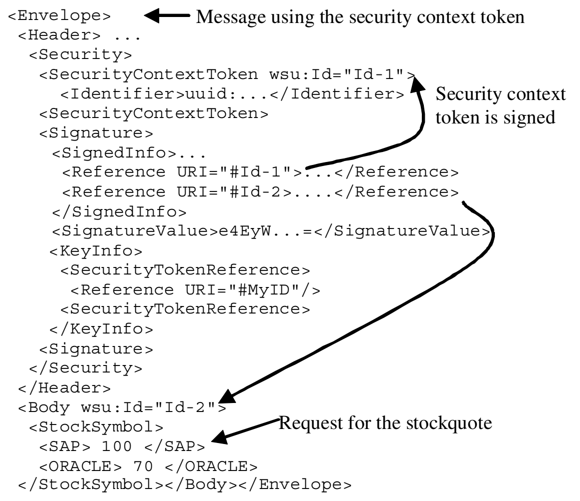
thereby, explicit security tokens, secrets, or key material need not be exchanged for every single message exchange and thus increased efficiency, better scalability, and security of the subsequent exchanges is achieved. Note that, we do not loose any features of message level granularities for signing and encrypting message parts as specified in WS-Security. Figure 4 (without dashed boxes and arrows) shows the current web service security architecture having the multiple message conversation in particular.

3. XML Rewriting Attacks

The trust and secure conversation frameworks described in the previous sections primarily provide us with meaningful protocols for establishing trust using security tokens and security context respectively. All the models are based on the WS-Security which is itself a framework only. Due to the complex and inter dependent structure of the frameworks, configuration mistakes at deployment time are likely to happen, in the worst case leading to an unsecured exchange either in single or multiple message scenarios. Besides, there are considerable numbers of limitations of the usage of the frameworks [21] which may affect the secured message exchange and overall performance of the system as well. We show some scenarios where possible security attacks may occur while using trust and secure conversation frameworks. We assume the presence of a malicious attacker in the SOAP message path from a sender to a receiver. The attacker may be a part of the message processing nodes (e.g. Intermediaries, partners) or any outsider which can listen, forward, and update to the messages. However, we assume the initial sender and the ultimate receiver of the messages are trusted.

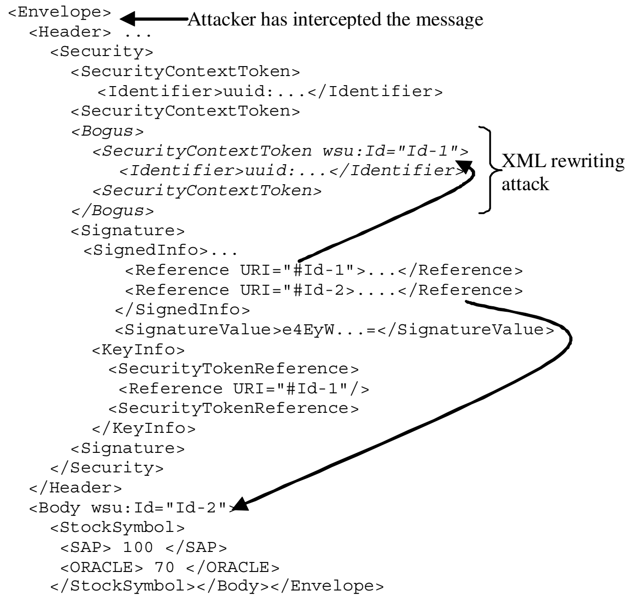
XML-Rewriting attacks [12] refer to a message modification by a malicious attacker while keeping the XML signature valid. An attacker may perform an attack on SOAP messages that use trust and secure conversation frameworks. We demonstrate two such attacks in the following.

Security Context Token Issuing. Requestor A wants to have a conversation with the service B. Requestor A requires a security context token which can be acquired from *STS* (Figure 4). *STS* essentially builds the trust between A and B. To be able to get a required security context token, A will send a SOAP message to the *STS* with a `RST` (`<RequestSecurityToken>` as Defined in WS-Trust) in the body of the message (see Figure 5). Since this is sensitive information, the `RST` element will be signed [8] by the requestor A. A may indulge in a conversation with service B, residing in different trust boundary, after receiving the security context token. Any malicious attacker in between the requestor A and the *STS* may capture the message and introduce its own `<BogusHeader>` inside the header of the SOAP message and copy the sensitive request information into the `<BogusHeader>` (Figure 6). Note that, the attacker does not change or modify any sensitive information and thus keeps the signature value intact for the *STS*. In


Message using the security context token
Security context token is signed
Request for the stockquote

```
<Envelope>
 <Header> ...
  <Security>
   <SecurityContextToken wsu:Id="Id-1">
     <Identifier>uuid:...</Identifier>
   <SecurityContextToken>
   <Signature>
    <SignedInfo>...
     <Reference URI="#Id-1">...</Reference>
     <Reference URI="#Id-2>....</Reference>
    </SignedInfo>
    <SignatureValue>e4EyW...=</SignatureValue>
    <KeyInfo>
     <SecurityTokenReference>
      <Reference URI="#MyID"/>
     <SecurityTokenReference>
    </KeyInfo>
   <Signature>
  </Security>
 </Header>
 <Body wsu:Id="Id-2">
  <StockSymbol>
   <SAP> 100 </SAP>
   <ORACLE> 70 </ORACLE>
 </StockSymbol></Body></Envelope>
```

Figure 7. Usage of Security Context Token (excerpt)



```
<Envelope>
  <Header> ...
    <Security>
      <SecurityContextToken>
         <Identifier>uuid:...</Identifier>
      <SecurityContextToken>
      <Bogus>
        <SecurityContextToken wsu:Id="Id-1">
           <Identifier>uuid:...</Identifier>
        <SecurityContextToken>
      </Bogus>
      <Signature>
       <SignedInfo>...
           <Reference URI="#Id-1">...</Reference>
           <Reference URI="#Id-2>....</Reference>
          </SignedInfo>
          <SignatureValue>e4EyW...=</SignatureValue>
        <KeyInfo>
          <SecurityTokenReference>
           <Reference URI="#Id-1"/>
          <SecurityTokenReference>
        </KeyInfo>
      <Signature>
    </Security>
  </Header>
  <Body wsu:Id="Id-2">
    <StockSymbol>
     <SAP> 100 </SAP>
     <ORACLE> 70 </ORACLE>
    </StockSymbol></Body></Envelope>
```

Figure 8. An attack on Usage of Security Context Token (excerpt)

addition, the attacker is adding its own request for a custom token (http://example.org/myBogusToken) and request type(`http://schemas.xmlsoap.org/ws/2004/04/security/trust/Renew`) into the body of the message. The *STS* may process the request by renewing the custom token assuming that the token has been established beforehand. Thus the attacker may posses a security token after the arrival of the `RSTR` token in response from the *STS*. The *STS* may ignore or reject the request but this attack enables the attacker to make an invalid request to the *STS*. Note that this kind of invalid request may occur for indefinite time and it could be resulted to a denial of service for the *STS*.

Similar kinds of attacks may occur for token renewing, validating or trust brokering scenarios. The consequences can be renewing request for an invalid security token, validating request for an invalid security token, and even brokering an invalid security token among the partners in a collaborative environment.

Usage of Security Context token. Requestor A wants to communicate with the service B in a conversation using the security context token that has been established using the SOAP request in Figure 5. Considering the performance bottlenecks of the usage of WS-Security features for every single message in the conversation, A establishes a security context token which is valid for the entire conversation using WS-Trust. Ideally, after the establishment of the security context token, A can continue in a conversation where each message will be secured using derived session keys. A can compute session keys using the shared secret of the security context token rather than exchanging the keys for every message. WS-SecureConversation specifies a security token to be used for the context authentication model named `<SecurityContextToken>`. Eventually, requestor A will use the established security context token for the later conversation with the service B. Service B provides stock information of various stock exchanges. Figure 7 shows one possible use of the established token to get stock information of SAP and ORACLE from service B. A signs [8] the established security context token and the body of the message, considering those elements as sensitive information. Any malicious attacker in between the requestor and the service may capture the message (Figure 8). The attacker may introduce an element `<Bogus>` of its own inside the `<Security>` header; keeping the signed security context token into it so that signature value is still valid. Now, an attacker may add its intended security context token which may or may not be valid to the service. The service may ignore or reject the attacker's security context token depending on its own policy. In case of an invalid security context token the service may reject the message according to its own policy. In case of arrival of an indefinite number of invalid requests the service may lead to a break down state. However, at any time several security context tokens may be valid between any two services for different conversations. An attacker may use any valid context token other than the intended one for the particular conversation which may eventually force the service to perform unnecessary computing tasks.

Similar kinds of attacks may occur for establishing, amending security contexts and computing and passing derived keys and session keys. The consequences would be as follows (not exhaustive): establishing invalid security context token with the attacker, amending any invalid security context token and thus invalid session or derived keys may be computed.

4. SOAP Account

A `SOAP Account` [21] describes the general idea of keeping a record of a SOAP message's structure of elements (e.g. Number of header elements, number of signed objects, and hierarchy information of the signed object).

The idea of a `SOAP Account` originates from the exploitation of an attacker against SOAP messages which is essentially a message restructuring effort, resulting in XML rewriting attacks [12]. A solution for detecting such attacks should comply with the standards that are widely accepted; should not violate XML usage motivation; and should be better performing. While using WS* policy we could also detect XML rewriting attacks, this is quite resource demanding and the setup is reasonably complex leading to errors. We take the `SOAP Account` approach which allows us to detect XML rewriting attacks early in the validation process. Besides, the usage of a `SOAP Account` is simple enough in a conversation of SOAP messages. The compliance with the standard and simplicity of this approach makes it less vulnerable and the deployment requires less effort thereby.

We believe that usage of `SOAP Account` in the web service architecture as shown in the Figure 4 (with dashed boxes) is capable of detecting any XML rewriting attack with improved performance compared to a policy based solution. The difference between previous architecture and the proposed one in Figure 4 is incorporating the `SOAP Account` approach. Ideally every SOAP processing node (e.g. sender, intermediary and receiver) should incorporate a `SOAP Account` data structure into its processing. The sending side will add required structure information into the `SOAP Account` before sending the message. The receiver will compute the `SOAP Account` information of the signed part in the received message and will compare this computed `SOAP Account` information with the attached `SOAP Account` information in the received message. If there is a difference between the computed value and the attached `SOAP Account` information, then we can conclude that there is a XML rewriting attack. In the later sections we provide performance analysis related data which supports our claim. `SOAP Account` information will be signed by the sender since it is sensitive information. However, an attacker might exploit the technique of XML rewriting attack on `SOAP Account` itself. Our previous work [16] describes how we can address and solve this issue.

Detecting XML Rewriting Attacks: At the time of sending SOAP messages either in a trust scenario and/or secure conversation scenario, we can always keep an account of structure of the SOAP elements by including the following information into the `SOAP Account` header (not exhaustive):

- Number of child elements of the root (Envelope).
- Number of header elements.
- Number of references for signing element.
- Predecessor, Successor, and sibling relationship of the signed object.

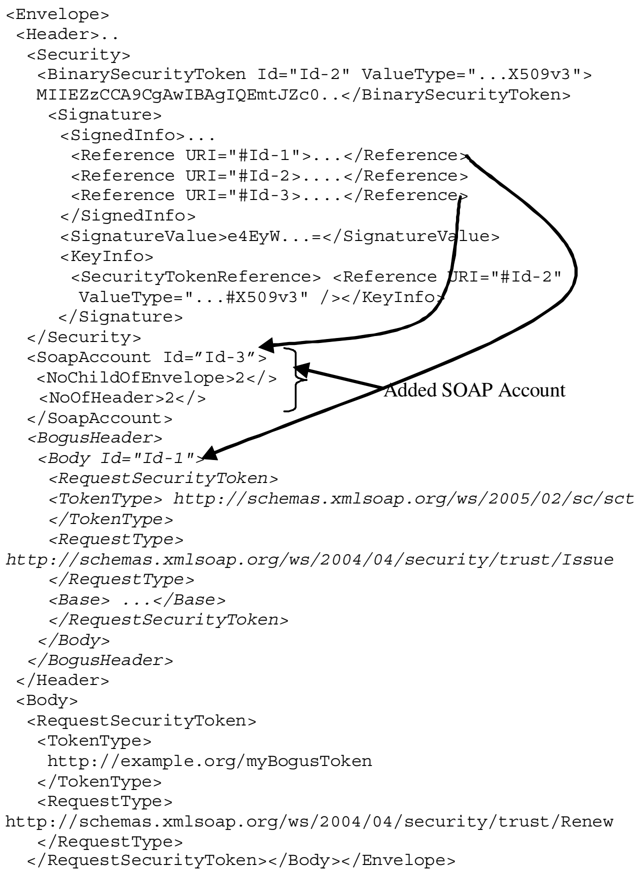
As a `SOAP Account` represents any kind of message structure information this list is not exhaustive rather it is always extensible depending on the context. For example, one might add the depth information of the signed element with respect to the root of the message (i.e. Envelope) to capture the location of the element. Even though using the depth information of a signed element we can detect XML rewriting attacks it may not comply with the schema of the WS* standards or even it may violate further processing of XML messages. The `SOAP Account` information for the signed part is computed while we are creating the message itself in the sender side. We do not incur any considerable overheads for the computation [21]. Section 5 provides a more rigorous analysis regarding the performance issues of incorporating `SOAP Account`. We take the attacking scenarios of section 3 respectively and show how our `SOAP Account` can be used to detect those attacks.

SOAP Account in Security Context Token Issue. Requestor A attaches `SOAP Account` information into the message (Figure 9) before sending it to *STS*. We add only two structure information namely `<NoChildOfEnvelope>` and `<NoOfHeader>` to capture the number of child elements in the SOAP envelope and number of header elements in the SOAP header respectively. The `SOAP Account` is signed by A before sending the message. Figure 9 shows the message excerpt after an attempt to attack by the attacker in the same way described in the section 3. Any legitimate receiver (e.g. *STS*) of the message complying with the `SOAP Account` approach will compute the `SOAP Account` information in the received message as soon as it arrives. The computed `SOAP Account` in *STS* is as follows:

```
<SoapAccount>
<NoChildOfEnvelope>2</>
<NoOfHeader>3</></SoapAccount>
```

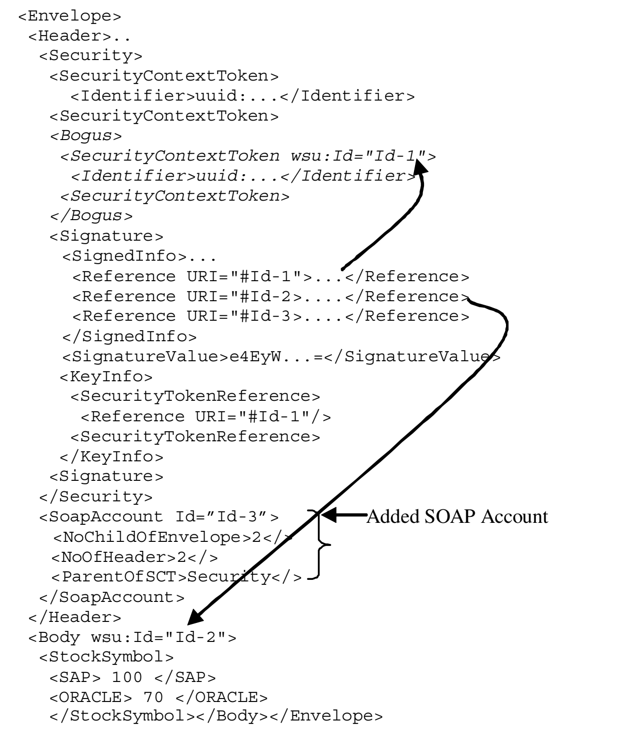
Note that, `<NoOfHeader>` is 3 including the `<BogusHeader>`. It does not match with the attached `SOAP Account` information as `<NoOfHeader>` contains 2. The receiver can immediately detect that there has been a rewriting attack on this message and rejects the request for issuing security context token. Note that the receiver can detect the attack by simple comparison which allows the early detection of rewriting attacks before signature validation and committing its resources to the request.

SOAP Account in Usage of Security Context Token. We take the same scenario of section 3, and as before we attach `SOAP Account` information into the message (Figure 10). However, we add different structure information to capture the parent information of a signed element namely, `<SecurityContextToken>`. In addition to the two previous structure information namely `<NoChildOfEnvelope>` and `<NoOfHeader>`, which capture the number of child elements in the SOAP envelope and number of header elements in the SOAP header respectively, we add this structure information. However, to detect this attack only one structure information in the `SOAP Account` namely `<ParentOfSCT>` would be sufficient. We keep the two previous structure information so that we can detect the attack described in Figure 9. The `SOAP Account` is signed by the sender before sending the message. Figure 10 shows the message excerpt after an attempt to attack by the attacker in the same way described in the



```
<Envelope>
 <Header>..
  <Security>
   <BinarySecurityToken Id="Id-2" ValueType="...X509v3">
   MIIEZzCCA9CgAwIBAgIQEmtJZc0..</BinarySecurityToken>
    <Signature>
     <SignedInfo>...
      <Reference URI="#Id-1">...</Reference>
      <Reference URI="#Id-2>....</Reference>
      <Reference URI="#Id-3>....</Reference>
     </SignedInfo>
     <SignatureValue>e4EyW...=</SignatureValue>
     <KeyInfo>
      <SecurityTokenReference> <Reference URI="#Id-2"
       ValueType="...#X509v3" /></KeyInfo>
     </Signature>
  </Security>
  <SoapAccount Id="Id-3">
   <NoChildOfEnvelope>2</>
   <NoOfHeader>2</>
  </SoapAccount>
  <BogusHeader>
   <Body Id="Id-1">
    <RequestSecurityToken>
    <TokenType> http://schemas.xmlsoap.org/ws/2005/02/sc/sct
    </TokenType>
    <RequestType>
http://schemas.xmlsoap.org/ws/2004/04/security/trust/Issue
    </RequestType>
    <Base> ...</Base>
    </RequestSecurityToken>
   </Body>
  </BogusHeader>
 </Header>
 <Body>
  <RequestSecurityToken>
   <TokenType>
    http://example.org/myBogusToken
   </TokenType>
   <RequestType>
http://schemas.xmlsoap.org/ws/2004/04/security/trust/Renew
   </RequestType>
  </RequestSecurityToken></Body></Envelope>
```

Figure 9. A RST token with SOAP Account after an attempt to attack (excerpt)



```
<Envelope>
 <Header>..
  <Security>
   <SecurityContextToken>
     <Identifier>uuid:...</Identifier>
   <SecurityContextToken>
   <Bogus>
    <SecurityContextToken wsu:Id="Id-1">
     <Identifier>uuid:...</Identifier>
    <SecurityContextToken>
   </Bogus>
   <Signature>
    <SignedInfo>...
     <Reference URI="#Id-1">...</Reference>
     <Reference URI="#Id-2>....</Reference>
     <Reference URI="#Id-3>....</Reference>
    </SignedInfo>
    <SignatureValue>e4EyW...=</SignatureValue>
    <KeyInfo>
     <SecurityTokenReference>
      <Reference URI="#Id-1"/>
     <SecurityTokenReference>
    </KeyInfo>
   <Signature>
  </Security>
  <SoapAccount Id="Id-3">
   <NoChildOfEnvelope>2</>
   <NoOfHeader>2</>
   <ParentOfSCT>Security</>
  </SoapAccount>
 </Header>
 <Body wsu:Id="Id-2">
  <StockSymbol>
   <SAP> 100 </SAP>
   <ORACLE> 70 </ORACLE>
   </StockSymbol></Body></Envelope>
```

Figure 10. An attempt to attack on Usage of Security Context Token with SOAP Account (excerpt)

section 3. Any legitimate receiver of the message complying with the SOAP Account approach will compute the information in the received message as soon as it arrives. The computed SOAP Account is as follows:

```
<SoapAccount>
<NoChildOfEnvelope>2</>
<NoOfHeader>2</>
<ParentOfSCT>Bogus</></SoapAccount>
```

Note that, <ParentOfSCT> is Bogus after the attack. It does not match with the attached SOAP Account information as <ParentOfSCT> contains Security. The receiver can immediately detect that there has been a rewriting attack on this message and rejects the usage of the security context token. Again the receiver can detect the attack by simple comparison.

5. Performance Issues

The performance of detecting XML rewriting attacks using a SOAP Account in a single message scenario is discussed in [21] and [16]. This section now focuses on performance regarding our trust and conversation scenarios. To understand the performance issues of detecting rewriting attacks in the trust and conversation scenario we need to revisit the goals of WS-SecureConversation. The first two goals are to establish or amend a security context token which will be valid for the entire conversation. The last goal is the derivation of session keys from the established context. To achieve any goal we need to use RST and RSTR tokens in the SOAP messages. This token can be used in the body or in the header of the SOAP messages which means this token is vulnerable to XML rewriting attacks also. These attacks can be detected using SOAP Account immediately as we have seen in section 4. So, early detection of XML rewriting attacks in any SOAP message which aims at context establishment, amendment or derivation of keys is possible when we use SOAP Account.

As long as the security context is established, the partners (e.g. requestor, service) can interact with each other for the entire conversation. Please note that each message in the conversation is also vulnerable to XML rewriting attacks. We perform our performance evaluation of detecting XML rewriting attacks using a SOAP request for security context token as of Figure 5. The body of the SOAP request contains a RST element according to WS-Trust and WS-SecureConversation. We implement a local *STS* which can provide security context token for a secured SOAP request. We simulate an attacker which is capable of intercepting the request and perform a rewriting attack before sending its own forged request as of Figure 6. The requestor, enforced with SOAP Account, adds its SOAP Account information into the SOAP Account header and signs it as of Figure 9. In our implementation the requestor added only the number of header information into <Soap Account> header as it is sufficient to detect the attack of Figure 6. The service, in this case, the *STS* will check the SOAP Account information of the received soap request with the attached <Soap Account> header. We perform this simple simulation in our performance evaluation emphasizing on several performance criteria. We observe the results which are described in the following.

Assuming the security context token has been established without an attack, any further attempt to attack on the messages in the conversation will be detected in the receiving end. The reason is as follows: After the secure establishment of the security context each message in the conversation is supposed to use the established security context token as a shared secret or any derived key from the secret for signing/or encryption of the message parts. The attacker must not get the shared secret or any derived key unless the sender and the receiver are malicious one.

5.1. Environment

We chose the WSS4J API [3] of Apache to simulate our idea of a SOAP Account in Java 1.5.0_06. However, in [21] we have used XWS Security Framework [5] of Sun JWSDP to simulate the same in a single message exchange scenario. While both of these mentioned libraries provide WS-Security implementation, the former provides implementation for WS-Trust and Ws-SecureConversation. All our performance data is computed on a Pentium 4 with 1.6 GHz CPU and 448 MB RAM. We have performed the simulation on Windows XP Professional with SP2. The *STS* Service is deployed as a web service in Axis 1.4 [1]. For our simulation we require SOAP Account processing, policy processing, signature processing and an attacker which performs XML rewriting attacks. On the requestor side, we have designed individual classes to simulate each of the mentioned processing. In the *STS* side, we designed all processing modules as individual handlers [1]. Note that, we simulate the Attacker as a handler in *STS* for the sake of simple processing. In practice, the attacker can reside anywhere in the SOAP message path. The requestor is deployed in the same computer where the *STS* is deployed. This way we avoid the network overhead.

We have obtained our simulation result with 100 iterations, but due to space limitations we show data with 50 iterations. In the first iteration the input SOAP message's size is 2695 bytes. We increase our input SOAP message size incrementally by 1 Kb in the consecutive iteration. The 1 Kb random XML elements has been added as a body element in every iteration. So the size of the request SOAP message in the 50th iteration is more than 50 Kb. Initially, in each iteration of a SOAP request, we performed several runs and took average computing time. However, the average computing time was comparable with a single running time in an iteration. Later on, we stick on computing time using a single run in each iteration. The Figure 11 shows a policy file that we use for policy enforcement in the requestor and the receiver. This simple policy file says that the body of the SOAP message should be signed. Note that, the enforcement of this simple policy file can detect the attack scenarios of Figure 6. In practice, the attacks may not be as simple as it is in the example scenarios, rather the attacker can perform XML rewriting on any part of the SOAP message.

5.2. Performance Criteria

Within a more general perspective our performance analysis is focused on two aspects: The enlargement of the SOAP message and computing time of various processing (e.g. Enforcement time, Signature Processing time, XML rewriting attack detection time). As a `SOAP Account` is essentially adding yet another header into the SOAP message we have to consider the overhead due to the enlarged SOAP header. However, to detect XML rewriting attacks in single message and conversation scenarios `SOAP Account` approach overrides the usage of a policy file. Any signature and encryption operation on XML message requires considerable XML processing time which is directly proportionate to the size of the message [15]. Our performance analysis will be based on the following criteria:

1. Relative comparison of request SOAP size vs. requestor `Soap Account` header size and Policy file size.
2. Relative comparison of `SOAP Account` size vs. Policy file size.
3. Relative comparison of signature processing in both ends.
4. Relative enforcement time of `SOAP Account` and Policy in sender (requestor).
5. Relative enforcement time of `SOAP Account` and Policy in the receiver.
6. Relative comparison of XML rewriting attack detection time using `SOAP Account` and Policy.

First, we perform a test with different sizes of SOAP messages starting with 2695 bytes up to 51551(∼50 Kb) bytes to observe the relative size of the `SOAP Account` header and Policy file (Figure 11) of the requestor. Figure 12 shows that both, a `SOAP Account` header and policy file are consuming considerably less space with compare to the request SOAP message. As a matter of fact, the size of the `SOAP Account` header and policy file remains the same throughout the 50 iterations with increasing SOAP messages. However, looking at the relative size of the `SOAP Account` and policy file we can conclude that the `SOAP Account` header consumes less space than the policy file. The size of the `SOAP Account` header and the policy file are 197 Kb and 388 Kb respectively. This implies that the usage of a SOAP Account does not consume any considerable memory space comparing to request Soap message and policy file. Here the `SOAP Account` consumes only ∼0.72% of the increasing request SOAP size whereas the policy file consumes ∼1.42% of the size of the request SOAP message. As we have message overhead data with respect to message size we further move on to an analysis of the required computing time. Figure 13 shows a comparative signature processing time on the requestor and the receiver side. It shows that the signature processing time on the requestor side is comparatively more than the receiver side. Digital signatures may require attaching time for the key or security token for verification, acquisition time of keys from the key-storage, signing time with

```
<wsp:Policy  ....> ....
        <!-- Example Message Policy -->
        <sp:SignedParts>
                <sp:Body/>
        </sp:SignedParts></wsp:Policy>
```

Figure 11. Policy file (excerpt)

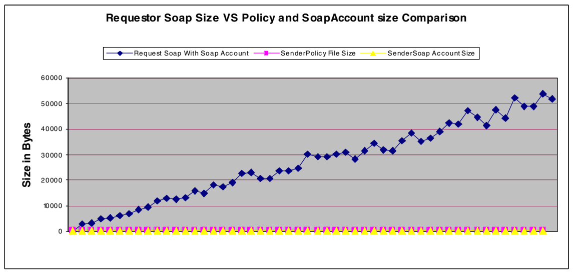


Figure 12. Requestor SOAP size vs Soap Account and Policy file size

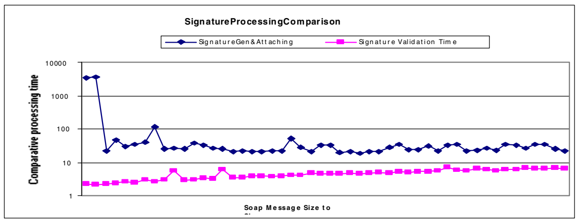


Figure 13. Requestor Signature processing vs Receiver Signature processing time

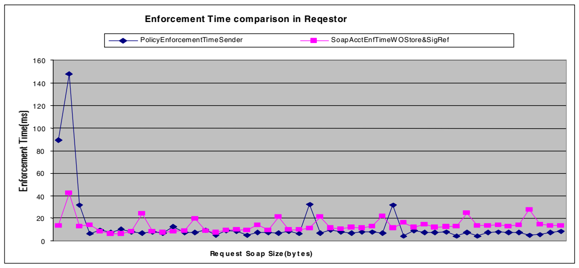


Figure 14. Requestor Soap Account enforcement vs Policy enforce time

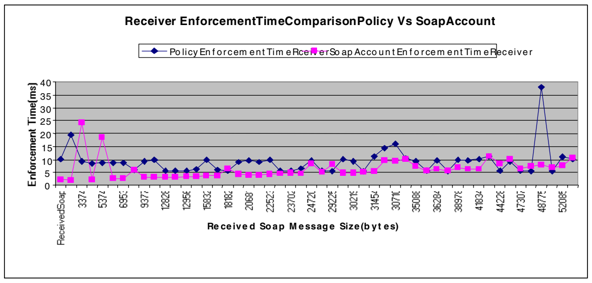


Figure 15. Receiver Soap Account enforcement vs Policy enforce time

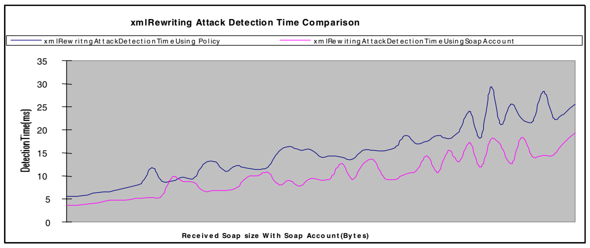


Figure 16. XML rewriting attack detection time Soap Account vs Policy

a particular algorithm and then attaching time of the signature value into the message. However, on the receiver side verification may not need all the processing that is required by the requestor. Initially, requestor may incur some overhead time due to JVM warm up time which is reflected in the initial spike in Figure 13. It shows that the requestor takes ∼15% more time to do the signature processing and attaching compared to the verification time of the receiver. To avoid the considerable signature processing time by the sender in general, we should be precise about the signed part of the message. Being precise about the signed part, allows us to avoid unnecessary signature processing on message parts which are not sensitive. This way we can preserve valuable computing time for signing only the required sensitive part. This also suggests that deciding what structure information will be attached into a `SOAP Account` affects the signature processing directly. Depending on the scenarios, only the related or required parts of the message should be in the signed part. In our example scenario of Figure 9 we demonstrate two structure information: `<NoChildOfEnvelope>` and `<NoOfHeader>`. However, at implementation time we attach the only latter due to the fact that the number of header information is enough to detect the XML rewriting attack in the Figure 6.

Figure 14 shows a comparative result of enforcement time of `SOAP Account` and policy on the requestor side. During the enforcement of `SOAP Account` the required structure information is computed and attached into the `SOAP Account` header. During the enforcement of policy the required signed parts, encryption parts are computed and security tokens are attached into the SOAP message according to the policy. The elapsed time in the enforcement for the both techniques is comparable. Again, in the beginning of the simulation there are some irregularities due to JVM warm up time. If we observe the initial spikes more closely we see that policy enforcement time is considerably higher than the `SOAP Account` enforcement time. This is because the requestor has to fetch the policy file from the storage which incurs computing time in addition to the JVM warm up time. As soon as the policy file is fetched, the later iterations may not need to fetch the file again but rather enforce it only. This result complements our previous observation of attaching required structure information so that enforcement time of `SOAP Account` in the requestor side can be a minimum.

On the receiver side, enforcement time of `SOAP Accout` is considerably less than the enforcement time of policy. Figure 15 shows that using `SOAP Account` we can gain ∼0.30% enforcement time compared to using policy enforcement. The uneven high spike in the beginning is due to the JVM warm up time. However, looking at the curves after the SOAP size exceeding 4500 bytes we observe that the `SOAP Account` enforcement time is getting closer to policy enforcement time. This suggests that we should be more careful about choosing structure information in the `SOAP Account` as the SOAP message size is increasing.

Figure 16 shows the data in support of our claim that using a `SOAP Account` approach on the receiver side we can detect XML rewriting attack early in the validation process. If we observe Figure 16 closely, we see that a `SOAP Account` approach is always faster than the policy based approach in detection of XML rewriting attack. This simulation is also done with the same increasing SOAP request from 1Kb to more than 50 Kb. Unlike the other simulation results it does not show any irregularity while detecting the attack. A `SOAP Account` based approach is ∼1.50% faster than a policy based approach. It also scales with the increasing SOAP size as it always outperforms the policy based approach.

In general, a `SOAP Account` based approach can be used in a XML rewriting attack prone service where performance of the service is an issue. Performance of services is a demand in a collaborative environment. Our `SOAP Account` approach shows convincing results in detecting XML rewriting attacks. However, we should be careful enough about certain performance criteria, particularly, on the requestor side. We should be precise enough in attaching structure information into `SOAP Account`. Attaching unnecessary structure information may lead to a performance bottleneck in the enforcement in the requestor side. We should also use cryptographic operations (e.g. signature, encryption) only when it is required and we should be granular enough in selecting the parts of the message to be signed and encrypted. This allows us keeping computing intensive signature and encryption processing as low as possible.

6. Related Work

There has been a rapid progress in specifying web service standards, specifically in the secure web service area. Functional aspects and performance issues of security related web service standards are ongoing research challenges. Some work has been done regarding functional analysis of WS* standards, but none has been performed in the context of a collaborative environment. There are so far a considerable number of implementations of SOAP and XML security services, but there is comparatively little work on analyzing their performance issues. At the time of writing there was no performance analysis of the state of the art in detecting XML rewriting attacks and only a few more generally comparable approaches.

In [14], the authors describe semantics for WS-Trust and WS-SecureConversation and prove security properties based on the semantics. They mention several limitations of these specifications as well. In [10], the authors describe an implementation of WS-SecureConversation focusing on group communication in a grid environment. They provide a performance and vulnerability analysis of their implementation. [13] describes generating and analyzing web services security policies to detect XML rewriting attacks based on a formal approach. It also describes a formal semantics for WS-SecurityPolicy for specifying the security goals of web services and their clients.

In [9], the authors focus on XML parsing, signature, and encryption computing time using different algorithms. They demonstrate the proof of concept of WS-SecureConversation being efficient with compare to WS-Security. In [19], the authors analyze WS-Security and RMI-SSL as comparable technologies. They describe the functional differences between them and show performance differences related to the architecture and implementation. A similar performance analysis has been done in [11], but in the context of RMI tunneling techniques and web services in general.

7. Conclusion and Future Work

In a collaborative environment, different partners are residing in different trust boundaries. Often they communicate with each other using single message or in a conversation of messages. In this paper, we have described briefly the insights into the required trust framework and secure conversation framework that are required for such collaboration. We have emphasized on secure message exchanges among the partners in collaboration.We have shown two example scenarios demonstrating possible XML rewriting attacks and the usage of `SOAP Account` to detect those attacks. We compared our `SOAP Account` approach with the WS* policy based approach with a detailed performance analysis and precise performance criteria. Finally, based on the performance analysis we observed some useful results regarding the potential usage of `SOAP Account` and discussed the recommended usage of `SOAP Account` accordingly.

SOAP messaging is meant to be the future means of communication between heterogeneous environments. Ideally, we can, for example, invoke any service defined in .NET environment using a client in JAVA and vice versa. However, there are some interoperability issues for which services may not interoperate with each other efficiently. For example, SOAP encoding is such an interoperability issue among services. Apache services [2] and .NET services [4] use different SOAP encoding by default during service invocation and operation. We performed all our simulation in a JAVA environment. It would be interesting to see how the simulation results show up in heterogeneous environment.

We will now also start to investigate performance with respect to the surrounding organisational context. This will include identifying critical and context-dependent securing of single SOAP messages and parts of a conversation as opposed to securing the entire channel or conversation.

References

[1] Apache soap implementation axis, http://ws.apache.org/axis/.

[2] Apache web service project, http://ws.apache.org/.

[3] Apache ws-security implementation, http://ws.apache.org/wss4j/.

[4] Basics of .net, http://www.microsoft.com/net/basics.mspx.

[5] The java web service tutorial, http://java.sun.com/webservices/docs/1.6/tutorial/doc/. June 14 2005.

[6] T. N. Chris Kaler. Web services security policy language (ws-securitypolicy),version 1.0,18 december 2002, http://www.verisign.com/wss/wssecuritypolicy.pdf.

[7] G. K. A. L. N. M. H. F. N. S. T. D. W. Don Box, David Ehnebuske. W3c. soap version 1.1, http://www.w3.org/tr/soap/.

[8] D. S. Donald Eastlake, Joseph Reagle. Xml signature syntax and processing, http://www.w3.org/tr/xmldsig-core/.

[9] G. F. Hongbin Liu, Shrideep Pallickara. Performances of web service security, http:// grids.ucs.indiana.edu/ptliupages/publications/wssperf.pdf.

[10] M. P. S. P. Hongbin Liu, Geoffrey Fox. A multi-party implementation of ws-secureconversation.

[11] K. B. H. M. R. I. Juric, M. B. and I. Vezocnik. Java rmi, rmi tunneling and web services comparison and performance analysis. volume SIGPLAN Not, pages 58–65, May 2004.

[12] A. D. G. G. O. Karthikeyan Bhargavan, Cdric Fournet. An advisor for web services security policies. volume 2005 Workshop on Secure Web Services, pages 1–9, Fairfax, VA, USA, November 2005. ACM, ACM Press, New York, NY.

[13] A. G. Karthikeyan Bhargavan, Cedric Fournet. Verifying policy-based security for web services. volume 11th ACM Conference on Computer and Communications Security of *CCS04*, page 268277, October 2004.

[14] C. F. A. D. G. Karthikeyan Bhargavan, Ricardo Corin. Secure sessions for web services. volume ACM Workshop on Secure Web Service of *SWS '04*, pages 56–66, Fairfax, Virginia, October 2004. ACM Press, New York, NY.

[15] P. Kumar. Xml processing measurements using xpb4j, http://www.pankaj-k.net/xpb4j/docs/measurements-may30/measurements-may30-2002.html. May 30 2002.

[16] M. A. R. A. S. R. Maarten. Towards secure soap message exchange in a soa. volume 3rd ACM Workshop on Secure Web Services of *SWS '06*, pages 77–84, Alexandria, Virginia, USA, November 2006. ACM, ACM Press, New York, NY.

[17] A. N. Martin Gudgin. Web services secure conversation language (ws-secureconversation), http://specs.xmlsoap.org/ws/2005/02/sc/ws-secureconversation.pdf.

[18] A. N. Martin Gudgin. Web services trust language (ws-trust), http://specs.xmlsoap.org/ws/2005/02/trust/ws-trust.pdf.

[19] B. B. M. C. M. H. Matjaz B. Juric, Ivan Rozman. Comparison of performance of web services, ws-security, rmi, and rmi-ssl, http://www.semgrid.net/citation-before-2006.1/+++jss-2006-service.pdf.

[20] H.-B. M. Nadalin, Kaler. Services security: Soap message security 1.0 (ws-security 2004), oasis standard 200401.

[21] M. A. R. R. M. A. Schaad. An inline approach for secure soap requests and earlyvalidation. volume OWASP Europe Conference, Leuven, Belgium, May 2006. OWASP, OWASP AppSec Europe.

[22] J. Schlimmer. Web services policy framework (ws-policy),september,2004.