An Inline Approach for Secure SOAP Requests and Early Validation[1]

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

Abstract. Regarding the current status of message level security in Web Services, various standards like WS-Security along with WS-Policy play a central role. Although such standards are suitable for ensuring end-to-end message level security as opposed to point-to-point security, certain attacks such as XML rewriting may still occur. In addition the generation and validation of the key security mechanisms (e.g. signature) are always processor intensive tasks. Based on some real world scenarios we propose a scheme to include SOAP Structure information in outgoing SOAP messages and validate this information before policy driven validation in the receiving end. This allows us to detect some XML rewriting attacks early in the validation process, with an improved performance. We report on this efficient technique and provide a performance evaluation. We also provide insights into the WS-Security, WS-Policy and related standards' features and weaknesses.

Keywords: Web Services, WS-Security, XML rewriting attack.

1 Introduction

Web service specifications (WS*) have been designed with the aim of being composable to provide a rich set of tools for secure, reliable, and/or transacted web services. Due to the flexibility of SOAP-level security [1] mechanisms, web services may be vulnerable to a distinct class of attacks based on the malicious interception, manipulation, and transmission of SOAP messages, which are referred to as XML rewriting attacks [2]. Although WS-Security, WS-Policy and other related standards theoretically can prevent XML rewriting attacks in practice, incorrect use of these standards may make web services vulnerable to XML rewriting attacks.

All WS* security related specifications, however, introduce new headers in SOAP messages. So concerns about the operational performance of Web services security are legitimate because added XML security elements not only make use of more network bandwidth but also demand additional CPU cycles at both the sender side and at the receiver side. Therefore it is desirable to examine the performance issue of Web services security.

The main achievements of this paper are that we explore XML rewriting attacks [2] against web services. We propose measures detecting these attacks build on the idea of adding additional SOAP structure information. We further evaluate the performance of the proposed solution against the existing state of the art.

The paper is organized as follows. Section 2 discusses related work. Section 3 reviews the state of the art and its limitations. Section 4 illustrates attacker and rewriting attack sce-

[1]This work was funded in part by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project.
This paper reflects only the author's views and the Community is not liable for any use that may be made of the information contained therein.

narios. Section 5 presents our efficient solution to prevent these attacks. In Section 6 we describe a case study in which we applied our approach. Section 7 evaluates the performance of the approach.

2 Related Work

Security protocols, described using web service specifications (WS*), are getting more complex day by day. Researchers are applying formal methods to verify and secure the protocols' and specifications' goals. As new vulnerabilities are exposed, these specifications continue to evolve. Microsoft's SAMOA [3] project is among the pioneer efforts where web services specifications are analyzed using rigorous formal techniques.

One of the focus areas of the SAMOA project is to identify common security vulnerabilities during security reviews of web services with policy-driven security [2]. The authors describe the design of an advisor for web services security configurations, the first tool both to identify such vulnerabilities automatically and to offer remedial advice. While their previous work [4] is successful to generate and analyze web services security policies to be aware for vulnerabilities to XML rewriting attacks, this tool is able to bridge the gap between formal analyses and implementation quite efficiently.

The mentioned previous work [4] describes a formal semantics for WS-SecurityPolicy, and proposes an abstract link language [6] for specifying the security goals of web services and their clients. They present the architecture and implementation of fully automatic tools to compile policy files from link specifications, and to verify by invoking a theorem prover [7] whether a set of policy files run by any number of senders and receivers correctly implements the goals of a link specification, in spite of active attackers. Note that policy-driven web services implementations are prone to the usual subtle vulnerabilities associated with cryptographic protocols; these tools help prevent such vulnerabilities, as policies can be verified when first compiled from link specifications, and also can be re-verified against their original goals after any modifications during deployment.

The assumptions with all these formalizations are that the attacker can compose messages, replay them, or decipher them only if it knows the right key, but cannot otherwise decrypt the encrypted messages. A severe limitation is that these formalizations do not model insider attacks: principals with shared keys are assumed well-behaved.

Some guidelines about performance evaluation of web services are specified in [8]. They address about the performance overheads of web services: the XML size, the calculation of the message size, the choice of the XML parser, the costs of the serialization and deserialzation, the costs of connection establishment and the overheads at the network level.

The performance of web service security in terms of its relationship in both the implemented cryptography in Java and the properties of input documents like the size and the complexity of structure are investigated in [9]. Different cryptography algorithms and various kinds of key references in XML are compared as well. They also categorize the web service security activities consisting of at least cryptography operations and XML processing. Cryptography operations are byte based, structure neutral and computation intensive. They conclude that XML processing, in particular XML canonicalization (page 4, Section 3.5) is the area where WS security performance should focus on.

In our later evaluation we take such performance issues into account as well.

3 State of the Art

In this section we describe different web services standards' insights and limitations related to web services security that are deployed widely in web services technologies.

3.1 WS-Security

The WS-Security [1] specification defines an end to end security framework that provides support for intermediary security processing. Message integrity is provided by using XML Signature [10] in conjunction with security tokens to ensure that messages are transmitted without modifications. Message confidentiality is granted by using XML Encryption in conjunction with security tokens to keep portions of SOAP messages confidential. WS-Security seeks to encapsulate the security interactions described above within a set of security Headers.

3.2 WS-Policy

WS-Policy is essentially a logical predicate on SOAP messages over base assertions, determining which message parts must be present, signed, or encrypted. A standard policy framework would make it possible for developers to express the policies of services in a machine-readable way. Web services infrastructure could be enhanced to understand certain policies and enforce them at runtime. The policy framework currently expressed by WS-Policy [11] requires the definition of policy "Assertions" (predicates) for each domain to which policy is to be applied. Three examples of specifications defining such Assertions have been published to date:

- *WS-PolicyAssertions* [12], defining some general-purpose Assertions,
- *WS-SecurityPolicy* [5], defining policy Assertions for WS-Security and other specifications that might cover the same message security space, and
- *WS-ReliabilityPolicy* [14], defining policy Assertions for WS-Reliability [23] and other specifications that might cover the same reliable messaging space.

3.3 WS-SecurityPolicy

WS-SecurityPolicy1.0 [5], built on the WS-Policy [11] and WSPolicyAssertion [12], is a declarative XML format for programming how web services implementations construct and check WS-Security headers. It expresses policy in terms of individual headers on individual messages. It defines two base assertions for integrity and confidentiality. The more recent version WS-SecurityPolicy1.1 [15] expresses policy in terms of higher-level message patterns.

3.4 XML-Signature

XML Digital Signature specification [10] specifies how to describe, attach and validate a digital signature using XML. The structure of a digital signature as currently defined within the specification is shown in Fig 1.

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

Fig. 1. Structure of XML Digital Signature

The Signature element is the primary construct of the XML digital signature specification. The signature is generated from a hash over the canonical form of the manifest, which can reference multiple XML documents. Canonicalization means to put a structure in a standard format that is generally used. The `<SignedInfo>` element is the manifest that is actually signed. This data is sequentially processed through several steps on the way to becoming signed. A concrete example using XML Signature is given in Fig 2.

3.5 XML Canonicalization

XML canonical form states that XML documents though being logically equivalent within an application context may differ in physical representations based on XML permissible syntactic changes. These changes, for example, can be attribute ordering, entity references or character encoding. Basically this is a process of applying standard algorithms to generate a physical representation of an XML document. In XML security, there is a standard mechanism to produce an identical input to the digestion procedure prior to both signature generation and signature verification. Given its necessity, the speed of canonicalization will have an impact on the overall performance of SOAP security.

3.6 Limitations of the State of the Art

Considering the state of the art, various web service specifications [1],[10],[15],[17] and articles [22], we observe the following limitations to their applicability in secure web services:

1. It is not realistic to capture all security needs within a simple declarative syntax (e.g. WS-Policy, WS-SecurityPolicy).
2. In order to handle each Assertion, a policy processor must incorporate a domain-specific code module that understands the interpretation of that Assertion as defined in the domain-specific specification.
3. The interpretation is subject to human error, so without strict conformance tests, different implementations of the processor for each assertion may not be consistent.
4. Policy files need to be validated on application startup because if a policy file is compromised then malicious SOAP messages could be transported.
5. Enforcing the policy is totally domain dependent and it is a must. The strongest policy may be useless if it is not applied to the right message.
6. To enforce the policy for intermediaries is yet to be standardized.
7. Lack of standardization to retrieve policy for sender or receiver.
8. The digital signature references message parts by their Id attributes but says nothing of their location in the message. So an attacker can rewrite the message part placing it in a new header keeping the reference valid.
9. The message identifier is optional according to WS-Addressing [16] but must be present in a request if a reply is expected.

All these limitations may directly affect the security and the performance of the web services.

4 ATTACKER SCENARIOS

The presence of a hostile opponent who can observe all the network traffic and is able to send any fraudulent messages meeting the protocol requirements must always be assumed. So SOAP messages are vulnerable to a distinct class of attacks by an attacker placed in between any two legitimate SOAP nodes (Sender, Intermediaries, and Ultimate Receiver). The attacker intercepts the message, manipulates it and may transmit it. These kinds of attacks are called XML rewriting attacks [2].

4.1 Attack Patterns

We observe that XML rewriting attacks follow two patterns in general. The patterns give us indication about the security loophole. The general patterns are as follows:

1. **SOAP Extensibility Exploitation**: The attacker generates new SOAP elements and adds those into the message, keeping it well formed. Consequently malicious data may be transported.
2. **Copy & Paste:** The attacker copies parts of a message into other parts of that message, or into completely new messages which may be generated using the previous pattern.

The attacker is able to forge message parts and tricks the recipient in a way that it is impossible to detect any tampering if the standards (e.g. WS-Security, WS-Policy, WS-SecurityPolicy) are not used carefully. The important observation here is that not the cryptographic technique used as part of the standard has been broken but that the SOAP message structure has been exploited by an attacker. From a general perspective XML rewriting attacks exploit a known weakness of XML signatures that is described in section 3.6,item 8.

4.2 XML Rewriting Attacks

In this section, we see two concrete examples and show two errors that lead to typical XML rewriting attacks.

First Scenario: Consider, one service consumer of a Stock Quote service requests for some particular Stock (Fig 2). Each request causes the consumer to pay. We assume that one SOAP node (Ultimate receiver) is supposed to process the SOAP header or Body. The **<Security>** header block without a specified actor can be consumed by anyone, but must not be removed prior to the final destination. An attacker can now observe and manipulate the message on the SOAP path. He can move an element (e.g. **Message ID**) into a new, false header (e.g. **Bogu**s) (Fig 3); everything else, including the certificate and signature, remains same. The **<Bogus>** element and its contents are ignored by the recipient since this header is unknown, but the signature is still acceptable because the element at reference URI **"Id-1"** remains in the message and still has the same value. This may cause the consumer to pay several times for the same request and forces the service to do redundant work.

Second Scenario: A customer submits an order that contains an **orderID** header (Fig 4) through his mobile device. He signs the **orderID** header and the body of the request (the contents of the order). When this is received by the order processing sub-system (an intermediary), it may insert **shippingID** into the header. The order sub-system would then sign, at a minimum, the **orderID** and the **shippingID**, and possibly the body as well. Then when this order is processed and shipped by the shipping department, a **shippedInfo** header might be appended. The shipping department would sign, at a minimum, the **shippedInfo** and the **shippingID** and possibly the body and forward the message to the billing department for processing. The billing department can verify the signatures and determine a valid chain of trust for the order, as well as who did what. An attacker with access to any SOAP node can copy & paste the **orderID** in a bogus header which causes the Order Processing System to process the same request several times. The attacker can copy & paste the body of the message under a new fake header and may insert arbitrary order information to be processed subsequently.

4.3 State of the Art Approach Against Attack

Methodical usage of WS-Policy [11], WS-PolicyAssertion [12], and WS-SecurityPolicy [5] resists these attacks. Microsoft has performed some experiments on its Web Services Enhancement [21] tool using another tool called WSE Policy Advisor [2]. The later tool uses three assertions: the message predicate assertion from WS-PolicyAssertion, and the integrity and confidentiality assertions from WS-SecurityPolicy.

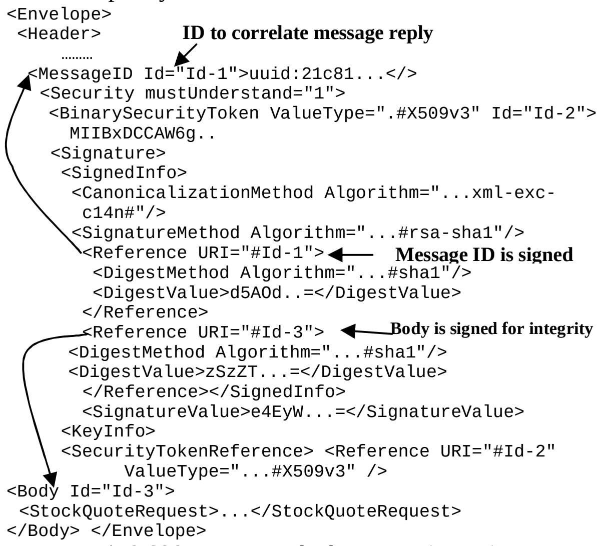
- A `<MessagePredicate>` assertion lists the message parts that are mandatory.

- An <`Integrity`> assertion lists the message parts to be jointly signed. The listed message parts must be signed if present.
- A <`Confidentiality`> assertion lists the message parts that must be encrypted if present in the envelope.

The flawed policy that is used to generate and check the request messages for Fig 2 is shown in Fig 5. In the Fig 5, the service is relying on the presence of the `<MessageID>` header for replay protection but the header is not included in the `<MessagePredicate>` assertion. This allows the attacker to introduce a new header enabling the replay attack.

The policy file in Fig 6 could be used in the second scenario and it does not include `<To>/<Action>` header in the `<MessagePredicate>` which enables attacker to route the message to another shop.

To check these policy errors WSE Policy Advisor includes 36 static queries [2]. Those queries enable us to detect the flaw. In this case the query is: “The Header`(MessageId,Header(To)`and `Header(Action)` are included in a message predicate assertion whenever the same header is included in an integrity assertion”. Note that after writing the policy the client and the service need to enforce it on all request or response messages. The enforcement is totally domain dependent. WSE does this using some mappings which are included in policy file.



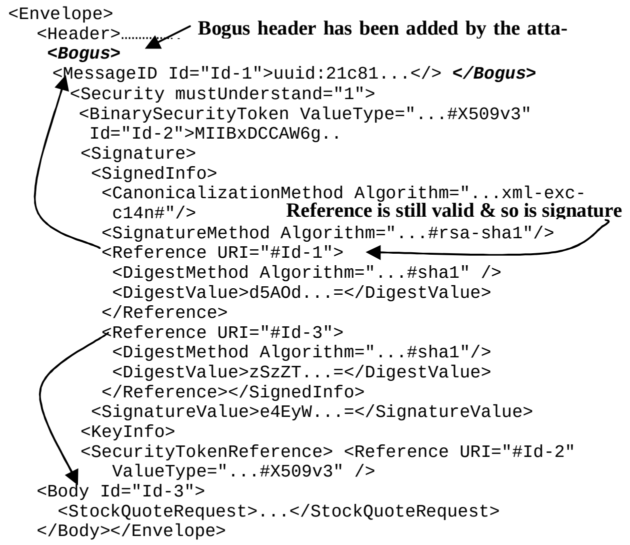
```
<Envelope>
 <Header>
    ………
  <MessageID Id="Id-1">uuid:21c81...</>
   <Security mustUnderstand="1">
    <BinarySecurityToken ValueType=".#X509v3" Id="Id-2">
      MIIBxDCCAW6g..
    <Signature>
     <SignedInfo>
      <CanonicalizationMethod Algorithm="...xml-exc-
       c14n#"/>
      <SignatureMethod Algorithm="...#rsa-sha1"/>
       <Reference URI="#Id-1">
        <DigestMethod Algorithm="...#sha1"/>
        <DigestValue>d5AOd..=</DigestValue>
       </Reference>
       <Reference URI="#Id-3">
      <DigestMethod Algorithm="...#sha1"/>
      <DigestValue>zSzZT...=</DigestValue>
       </Reference></SignedInfo>
       <SignatureValue>e4EyW...=</SignatureValue>
     <KeyInfo>
     <SecurityTokenReference> <Reference URI="#Id-2"
           ValueType="...#X509v3" />
<Body Id="Id-3">
 <StockQuoteRequest>...</StockQuoteRequest>
</Body> </Envelope>
```

Fig. 2. SOAP message sent by the requester (Excerpt)

5 INLINE APPROACH

Considering the specifications of WS-Security, WS-Policy, WS-SecurePolicy and above discussion, we observe that a large number of SOAP extensions is possible. SOAP header (`<Security>`) information never considers the SOAP message structure which is essentially the major attack point. In the context of our paper we recognize the dynamic structure of SOAP messages by referring to them using the term SOAP Account. We show that including SOAP Account information in SOAP we can detect these XML rewriting attacks early in the validation process. This SOAP structure (SOAP Account) information can be integrated easily while adding the headers. We need an inline approach to incorporate SOAP Account information into the message. The objectives of the proposed technique are:

1. To be able to pass SOAP Account information about the exchanged SOAP messages in a domain independent fashion.
2. SOAP messages should be protected while they are processed by a node such that they can be updated legitimately by the intermediaries if they are required to do so. Any tampering with pre-existing message parts by the compromising node must be detected early by the recipient before committing its resources to validate and to process the request.



```
<Envelope>
   <Header>
    <Bogus>
     <MessageID Id="Id-1">uuid:21c81...</> </Bogus>
      <Security mustUnderstand="1">
       <BinarySecurityToken ValueType="...#X509v3"
        Id="Id-2">MIIBxDCCAW6g..
       <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-3">
          <DigestMethod Algorithm="...#sha1"/>
          <DigestValue>zSzZT...=</DigestValue>
         </Reference></SignedInfo>
        <SignatureValue>e4EyW...=</SignatureValue>
       <KeyInfo>
       <SecurityTokenReference> <Reference URI="#Id-2"
          ValueType="...#X509v3" />
   <Body Id="Id-3">
     <StockQuoteRequest>...</StockQuoteRequest>
   </Body></Envelope>
```

Bogus header has been added by the atta-
Reference is still valid & so is signature

Fig. 3. SOAP message after attack (Excerpt)

The general objective is to protect the integrity features of a SOAP message while in transit from malicious attackers. While securing the requester and the service using WS-policy is well understood, securing the intermediaries using the same is not. A SOAP message may choose its next hop dynamically and its' required message parts may be secured dynamically based on the requirements of the intermediaries. We assume that the sender and the ultimate recipient of the message are always trusted.

5.1 Motivation of Using SOAP Account Information

After carefully observing the rewriting attacks, the following conclusions are obvious:

- All attacks are some kind of modification of a SOAP message (either deleting some parts and adding afterwards, or adding some completely new element in a SOAP message essentially in the Header portion or in Body).
- When some unexpected modification occurs in the form of manipulation of underlying XML elements, the intended predecessor or successor relationship of the SOAP element is lost consequently.
- The number of predecessor, successor, and sibling elements of a SOAP element where the unexpected modification occurs is changed and thus the expected hierarchy of the element is modified as well.

At the time of sending SOAP message, we can always keep an account of the SOAP elements by including SOAP Account into the message: (Fig 7)

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

These SOAP Account information are computed while we are creating the message itself in the sending application. We do not incur any considerable overheads for the computation. Since this information is computed while creating the same message we call it the *inline approach* in this paper.

5.2 Proposed Technique

On the sender side, the protocol stack generates SOAP envelopes that satisfy its policy and then we add SOAP Structure/Account information (Fig 7) into the outgoing SOAP message. The sender must sign the SOAP Account information.

Conversely, on the receiver side, a SOAP envelope is accepted as valid, and passed to the application; if its SOAP Account/Structure Information is validated at first and then policy is satisfied for this envelope. Validating SOAP Account information before validating policy, we can detect rewriting attacks in the first phase and thus without doing processing intensive policy driven validation which may not detect attacks at all if not used carefully (Fig 5,Fig 6).

5.3 Using SOAP Account Information in SOAP Message

Fig 7 depicts the SOAP Structure/Account Information that we consider in our implementation of Java SOAP Account module. We capture the SOAP structure that is computable using any SOAP processor. Using this information we are able to detect the attacks in the scenarios. We have left one extension element to include any future extension. In later sections we give a concrete example how can we add and validate SOAP Account information.

Before sending any SOAP message we calculate the SOAP Account information and capture it in SOAP Header Elements. We add this information in a SOAP element either in Header or in Body and then sign it (Preferably in Header). The same arguments are applicable for intermediaries as well. This calculation is done by the `AddSoapAccount` module in our implementation. To understand it rigorously we take the second scenario from Fig 4 and use the following notation:

Encryption of a plaintext m into a cipher text is written as C=$\{m\}_{K,}$ where K is the key being used.

A *digital signature* written as encryption $\{m\}_S^{-1}$, with a private signing key S^{-1} .When A sends some message m to B we write $A \rightarrow B : m$. We write $A \rightarrow B : \{m\}_S^{-1}$ when m is sent with a signature. Concatenation of m_1 and m_2 is written as m_1+ m_2. We define a *signed object pattern(SOP)* which manifests the signed elements in a message one intents to sign.

In Fig 8, A's *signed object pattern(SOP_A)* is {SOAP Account $_A$ + OrderID + Body},

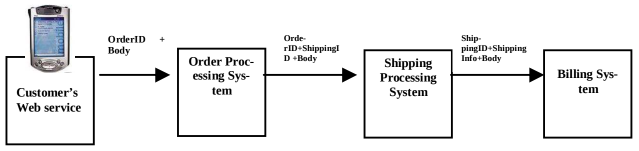

OrderID + Body
Orde-rID+ShippingID +Body
Ship-pingID+Shipping Info+Body
Customer's Web service
Order Processing System
Shipping Processing System
Billing System

Fig. 4. SOAP processing with multiple intermediaries

where SOAP Account$_A$ refers to the SOAP Account of A. A signs its *signed object pattern* before sending it to B:

A $\rightarrow$ *B :R, { SOAP Account$_A$* + *OrderID* + *Body }$_A^{-1}$*; Here R is the rest part of the message. B processes the order and adds new header ShippingID and B's *signed object pattern(SOP$_B$)* is {I$_1$ + SOAP Account$_B$ + ShippingID + Body}, where SOAP Account$_B$ refers to the SOAP Account of B.

```
<Policy Id="FlawedPolicy1">
 <MessagePredicate>
   Body() Header(To) Header(Action)
 </MessagePredicate>
 <Integrity>
  <TokenInfo>
   <SecurityToken>
    <TokenType>...#X509v3
    </TokenType>
    <TokenIssuer>CN=Root Agency
    </TokenIssuer>
   </SecurityToken>
  </TokenInfo>
  <MessageParts>
   Body() Header(To) Header(Action)
   Header(MessageID) Timestamp()
  </MessageParts>
 </Integrity>
</Policy>
```

MessageID is not included in the MessagePredicate

Fig. 5. FlawedPolicy1 (Excerpt)

```
<Policy Id="FlawedPolicy2">
 <MessagePredicate>
   Body() Header(MessageID)
 </MessagePredicate>
 <Integrity>
   <TokenInfo>
    <SecurityToken>
     <TokenType>...#X509v3
     </TokenType>
     <TokenIssuer>CN=Root Agency
     </TokenIssuer>
    </SecurityToken>
   </TokenInfo>
  <MessageParts>
   Body() Header(To) Header(Action)
   Header(MessageID) Timestamp()
   </MessageParts>
 </Integrity>
</Policy>
```

To/Action are not included in the MessagePredicate

Fig. 6. FlawedPolicy2 (Excerpt)

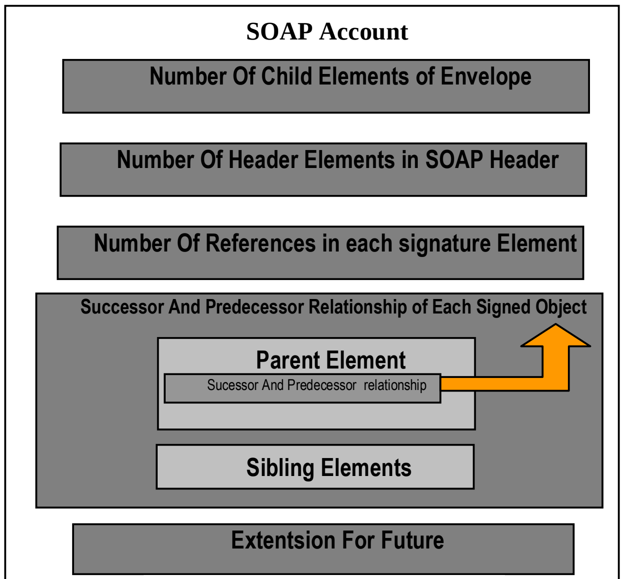

SOAP Account
Number Of Child Elements of Envelope
Number Of Header Elements in SOAP Header
Number Of References in each signature Element
Successor And Predecessor Relationship of Each Signed Object
Parent Element
Sucessor And Predecessor relationship
Sibling Elements
Extentsion For Future

Fig 7: SOAP Account

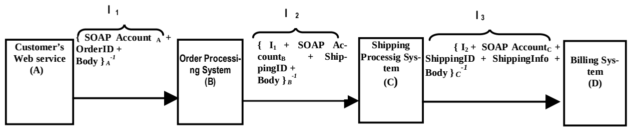

Customer's Web service (A)
I 1
{ SOAP Account A + OrderID + Body }A-1
Order Processing System (B)
I 2
{ I1 + SOAP AccountB + ShippingID + Body }B-1
Shipping Processig System (C)
I3
{ I2 + SOAP AccountC + ShippingID + ShippingInfo + Body }C-1
Billing System (D)

Fig 8: SOAP Processing with multiple intermediaries

$B \rightarrow C : R, \{$ I_1 + SOAP Account_B + ShippingID + Body $\}_{B^{-1}}$;Here I_1 is the signature of A's signed object pattern. C processes the order and adds new header ShippingInfo and C's *signed object pattern(SOP_C)* is {I_2 + SOAP Account_C + ShippingID + ShippingInfo + Body}, where SOAP Account_C refers to the SOAP Account of C.

$C \rightarrow D : R, \{$ I_2 + SOAP Account_C + ShippingID + ShippingInfo + Body $\}_{C^{-1}}$; Here I_2 is the signature of B's signed object pattern.

Finally, D receives: R,$\{\{\{SOP_A\}_{\boldsymbol{A}^{-1}} + \{SOP_B\}\}_{\boldsymbol{B}^{-1}} + \{SOP_C\}\}_{\boldsymbol{C}^{-1}}$. D will validate SOAP Account_C using `CheckSoapAccount` module in our implementation. D uses SOAP Account_C as C is the outermost signature, it needs to validate. Having a nested signature, D can validate each signature subsequently using each public certificate respectively. Note that all SOAP Account information is also well protected by a signature which makes it impossible to change by any malicious host. Now the virtue of SOAP Account will be manifested directly. If any kind of XML rewriting attack appears in the message in the form of the mentioned scenarios, it will be caught immediately by `CheckSoapAccount`. This is straightforward as each attack in a received SOAP message essentially invalidates the SOAP Account information that is bundled in the same SOAP message.

A key advantage of our approach compared to the policy-driven approach is that the deletion of headers and elements can be detected without restricting the flexible XML format. Deletion is a stronger form of a rewriting attack. In order to prevent this with the policy approach, every header and element should be declared as mandatory, which introduces first of all a performance penalty in the validation phase and more important it reduces strongly the flexibility of the XML message format. Only message elements that were defined in advance by the partners can be added. With the inline approach the different intermediaries still have some flexibility to add additional information, which can be detected for deletion/rewriting by the subsequent parties.

5.4 Where is the Efficiency?

One can argue about achieving efficiency in detecting XML rewriting attacks, considering the added modules (e.g. `AddSoapAccount, CheckSoapAccount`) at both ends. In Fig 8 we assume every SOAP node (A,B,C,D) has WS-security infrastructure implementation at least for signature generation (maybe for WS-Policy as well) and is supposed to process some headers. Every intermediary receives the message and it checks it in the `CheckSoapAccount` module before committing its resources for WS-Security and WS-Policy infrastructure. We show how we can detect the attacks in the added module whereas in the current approach the attacks may be undetected all the way to the ultimate end and even might be undetected in the end as well, unless it has a well designed policy. We show a concrete example describing this fact in the following section. We do a performance evaluation supporting this claim in section 7.

Note that the most popular XML packages comply with W3C Document Object Model (DOM), which provides a set of interfaces for creating, accessing and modifying both the structure and the content of the document. Security related XML processing includes parsing, validation, transformation and document tree traversal. As we mentioned in section 2 cryptographic processing (e.g. signing and verification, encryption and decryption, and among different algorithms) incurs negligible computing time where some researchers find that XML canonicalization is disproportionately time consuming. We can consciously ignore this XML canonicalization while validating SOAP Account information.

6 Case Study

6.1 Concrete Example

We consider a scenario where only one SOAP Account is attached with the requester's SOAP message and no intermediaries are supposed to update it. A customer, Alice, requests 1000 euros to be transferred from her account to the supplier (Bob's) account (Fig 9). Some malicious attacker intercepts this message and updates it stating to transfer 5000 euros instead of 1000 euros (Fig 10).

Fig 9 depicts the outgoing message after adding SOAP Account information. The policy file for this message would be Fig 11.

Observe that in spite of using `Body`() in `<MessageParts>` and `<MessagePredicate>` in Fig 11 the attack in Fig 10 is possible. This is due to the fact that Message Predicate only makes statements about mandatory parts of the message and the XML signature does not say anything about the location of the message parts to be signed as stated in section 3.6.

6.2 Adding SOAP Account Information into SOAP Message

Using the `AddSOAPAccount` module we can calculate any accounting information about the outgoing message in Fig 9. The SOAP Account of Fig 9 is as follows:

- Number of children of Envelope is 2
- Number of Header is 2
- Number of Signed Elements is 3
- Immediate Predecessor of the 1st Signed Element is "Envelope"
- Sibling Elements of the 1st Signed Element is "Header"

The Extension element of the SOAP Account (Fig 7) makes it easy to add any additional common required accounting information between sender and receiver. There should be an agreement about the SOAP Account information they require beforehand. This information is added into a header named "SoapAccount". Before sending the message, SOAP Account must be signed by the sender.

Generation of Soap Account information neither depends on any enforcement infrastructure nor does it incur considerable execution time. It is rather efficient in terms of execution time as a SOAP Account can be computed inline while generating the SOAP message. We can easily attach this information using existing SOAP message libraries which makes it robust.

6.3 Simulating the Attacker

To simulate the attacker in these scenarios we design a Java class `Attacker` which represents the malicious host. After receiving the message from the legitimate sender it updates the message following the attack patterns described in section 4.1 and sends the updated message to the next hop. But this attempt to attack is detected by the legitimate receiver of the message.

6.4 SOAP Account Validation

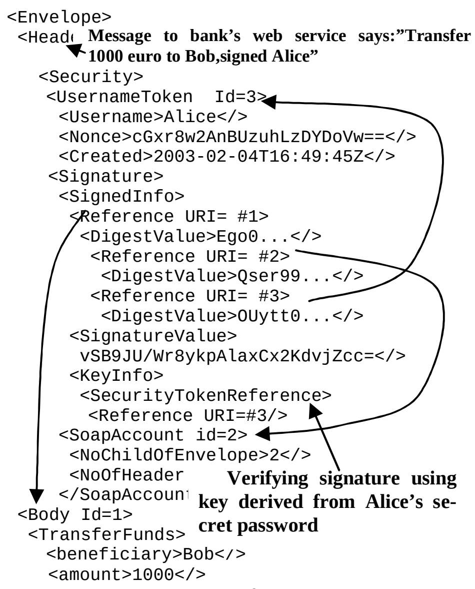
The Soap Account validation should be done as soon as the message is received, before doing any policy validation. The receiver calculates the SOAP Account information of the received SOAP message (Fig 10) using `CheckSoapAccount` module as follows:

- Number of children of Envelope is 2
- Number of Header is 3.
- Number of Signed Elements is 3

- Immediate Predecessor of the 1st Signed Element is “BogusHeader”
- Sibling Elements of the 1st Signed Element is “SoapAccount”,”Security”

On the other hand the obtained SOAP Account information as provided in the received SOAP message, (Fig 10) is as follows:

- Number of children of Envelope is 2
- Number of Header is 2.
- Number of Signed Elements is 3
- Immediate Predecessor of the 1st Signed Element is “Envelope”
- Sibling Elements of the 1st Signed Element is “Header”

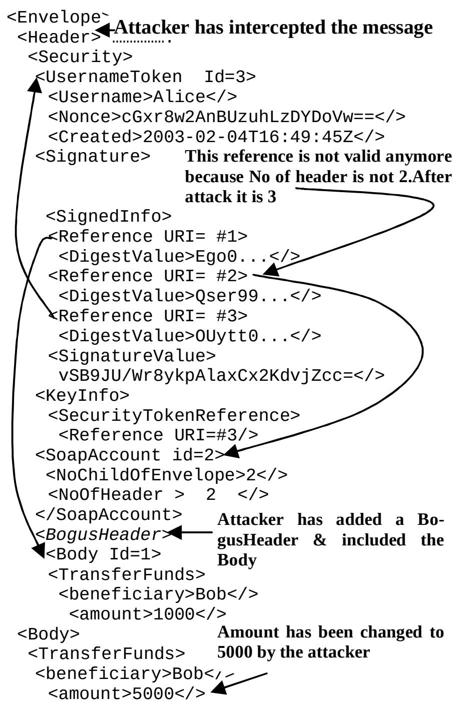


```
<Envelope>
 <Head
   <Security>
    <UsernameToken  Id=3>
     <Username>Alice</>
     <Nonce>cGxr8w2AnBUzuhLzDYDoVw==</>
     <Created>2003-02-04T16:49:45Z</>
    <Signature>
     <SignedInfo>
      <Reference URI= #1>
       <DigestValue>Ego0...</>
        <Reference URI= #2>
         <DigestValue>Qser99...</>
        <Reference URI= #3>
         <DigestValue>OUytt0...</>
      <SignatureValue>
       vSB9JU/Wr8ykpAlaxCx2KdvjZcc=</>
      <KeyInfo>
       <SecurityTokenReference>
        <Reference URI=#3/>
     <SoapAccount id=2>
      <NoChildOfEnvelope>2</>
      <NoOfHeader
     </SoapAccoun
 <Body Id=1>
  <TransferFunds>
    <beneficiary>Bob</>
    <amount>1000</>
```

Message to bank’s web service says:”Transfer 1000 euro to Bob,signed Alice”

Verifying signature using key derived from Alice’s secret password

Fig. 9. A SOAP request before an attack (Excerpt)



```
<Envelope>
 <Header>
  <Security>
   <UsernameToken  Id=3>
    <Username>Alice</>
    <Nonce>cGxr8w2AnBUzuhLzDYDoVw==</>
    <Created>2003-02-04T16:49:45Z</>
   <Signature>
    <SignedInfo>
    <Reference URI= #1>
     <DigestValue>Ego0...</>
    <Reference URI= #2>
     <DigestValue>Qser99...</>
    <Reference URI= #3>
     <DigestValue>OUytt0...</>
    <SignatureValue>
     vSB9JU/Wr8ykpAlaxCx2KdvjZcc=</>
   <KeyInfo>
    <SecurityTokenReference>
     <Reference URI=#3/>
   <SoapAccount id=2>
    <NoChildOfEnvelope>2</>
    <NoOfHeader >  2  </>
   </SoapAccount>
   <BogusHeader>
    <Body Id=1>
    <TransferFunds>
     <beneficiary>Bob</>
      <amount>1000</>
 <Body>
  <TransferFunds>
   <beneficiary>Bob</
    <amount>5000</>
```

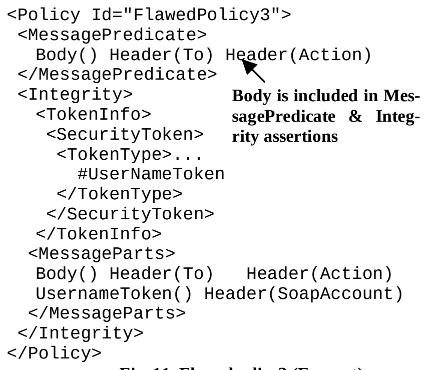
Attacker has intercepted the message

This reference is not valid anymore because No of header is not 2.After attack it is 3

Attacker has added a BogusHeader & included the Body

Amount has been changed to 5000 by the attacker

Fig. 10. SOAP request after an attempt to attack (Excerpt)



```
<Policy Id="FlawedPolicy3">
 <MessagePredicate>
   Body() Header(To) Header(Action)
 </MessagePredicate>
 <Integrity>
   <TokenInfo>
    <SecurityToken>
     <TokenType>...
       #UserNameToken
     </TokenType>
    </SecurityToken>
   </TokenInfo>
  <MessageParts>
   Body() Header(To)   Header(Action)
   UsernameToken() Header(SoapAccount)
  </MessageParts>
 </Integrity>
</Policy>
```

Body is included in MessagePredicate & Integrity assertions

Fig. 11. Flawedpolicy3 (Excerpt)

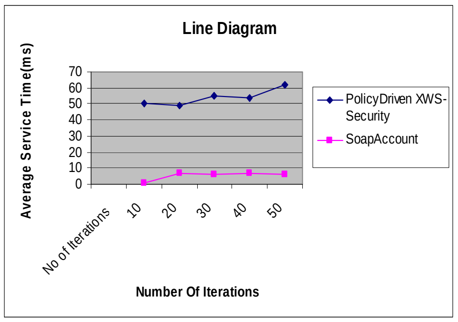


Fig. 12. Performance Diagram

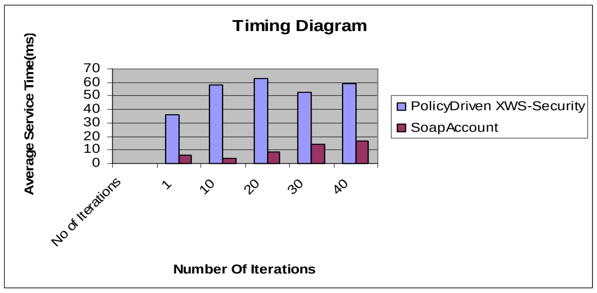


Fig. 13. Performance Diagram

If any mismatch happens the receiver can conclude that the SOAP message is not acceptable. In our proposed scenario, there is a clear mismatch. In addition, if an attacker changes the SOAP Account information meeting its updated SOAP message's account information, then this message will be invalidated in the receiving end while validating the signature of the signed SOAP Account by the initial sender. Again, no substantial execution time is required here as we can validate the SOAP Account information inline while reading the message. The performance evaluation in the next section describes this more in detail.

7 PERFORMANCE EVALUATION

Performance evaluation of Web services can help implementers understand the behavior of the services and gives an indication to the feasibility of the deployment. The most commonly used approach to obtain performance results of a given Web service is performance testing, which means to run tests to determine the performance of the service under specific application and workload conditions. To be more specific the total execution time of a process is a measure of its efficiency.

We use XWS-Security framework [13] as a comparable message level security infrastructure that has already wide deployments. XWS-Security framework has its own way of enforcing policy. XWS Configuration file [13] is a domain dependent way of enforcing policy in XWS-Security Framework in Java. This is essentially a XML file. Instead of using Policy directly this framework uses this XML file which has its own syntax and semantics for attaching and using the security features (e.g. attaching signature, referencing a key certificate).

Since we focus on the integrity aspect of a message which requires only signature infrastructure, we take the advantage of an already implemented signature infrastructure in the XWS-Security framework. In the process, we add Soap Account information in the outgoing message before incorporating WS-Policy in the message. This exception does not weaken our evaluation; rather it helps us to make a more solid claim. The XWS-Security framework which has both a signature and an encryption infrastructure, will incur more execution time in the sending side compared with the execution time of using the signature infrastructure alone. We show this in the following section.

7.1 Timing Analysis

We measure the execution time taken by a Web service invocation using two time frames: *Service Time (S)* and *Message Delay Time (M)*. Service Time is the time that the Web service takes to perform its task. In our case Service time is essentially the duration of detection of XML rewriting attacks. Message Delay Time is the time taken by the SOAP messages, in being sent/received by the invocation call. We simulate Message Delay using Random number to iterate a loop. It may be determined by the size of the SOAP message being transmitted/received and the load on the network through which the message is being sent/received. We do not consider the size of the message, as the same message is transmitted each time in the simulation to get the clear measurement and the load on the network is out of scope here. To be more specific, Message Delay time gets longer by the increased SOAP size of augmented SOAP headers but it is not a WS-Security specific concern.

Total Invocation Time (T) for a Web service σ is given by the following formula.

$$T(\sigma) = M(\sigma) + S(\sigma)$$

Evaluating the above two components of *T* for a Web service invocation, help us to analyze the efficiency of a Web service. We perform tests to determine each of the above two components for a number of iterations for a policy driven solution versus our proposal. We generate the SOAP message in Fig 9 in the sender side and we simulate an attacker as a malicious intermediary which generates the rewriting attacks as in Fig 10. We send the same message to the receiver for a specific number of iterations, while the attacker generates the same attack same number of times. Fig 12 and Fig 13 show corresponding charts in line and timing diagram for 50 and 40 iterations using different timing resolution of Java profiling. It is clearly indicative that the proposed method shows better execution time in comparison to the XWS-Security policy driven framework.

7.2 Evaluation Environment

The sender (e.g. `AddSOAPAccount`) and receiver code (e.g. `CheckSoapAccount`) are written in Java and they are compiled and executed with Sun's jdk1.5.0_06, for windows. To be more specific we use XWS Security Framework of JWSDP 1.6 package for WS-Security features. The experiments are carried out using a PC with Mobile Intel(R) Pentium(R) 4, 2.80GHz Processor and 512 MB RAM, running on Microsoft Windows XP Professional.

7.3 Discussion

The data in Fig 12 are extracted using the `System.currentTimeMillis()` Java method which has a resolution of 15/16 ms. The result shows an impressive performance against policy driven validation. In average, service time using SOAP Account is 10 times faster than using a comparable policy based approach. Fig 13 shows another performance diagram obtained using a library called "hrtlib.jar" [19] instead of using `System.currentTimeMillis()`, which improves the accuracy of 15/16 ms to a fraction of 1 ms(e.g. 0.5 ms).

As SOAP supports a variety of message exchange patterns, such as request-response, one way message, RPC, and peer-to-peer interaction, XML rewriting attacks are possible in any patterns. So is the SOAP Account driven validation to detect these attacks.

8 Conclusion and Future Work

SOAP structure information has been ignored in detecting XML rewriting attacks so far. We have presented and discussed an inline approach to include SOAP structure information (SOAP Account) in the SOAP message and to validate the information by the receiver of the message. SOAP Account information can be used to detect the XML rewriting attacks immediately in the receiving end which might not be detected using the state of the art (e.g. WS-Security, WS-Policy, WS-SecurityPolicy) as it is showed in the section 3. This simple and elegant feature can be incorporated in WS-Security. In particular we can attach SOAP Account information into `<Security>` header in the WS-Security. We can even use it in any standalone web service which may be subject to XML rewriting attacks. It is not an attempt to replace policy based validation; rather it is designed to be an alternative that can be used when performance is an issue in detecting XML rewriting attacks.

We have considered the SOAP structure information to be used in the context of securing single messages. Using WS-Security independently for each message to secure the integrity of a session of messages is rather inefficient. WS-SecureConversation [20] introduces security contexts, which can be used to secure sessions between two parties. How SOAP structure information can be used for securing a session is a future research topic. We have used only the XWS-Security Framework as a comparable message level security implementation and have drawn some comparisons of WSE implementation with our technique. It would be interesting to see how the performance scales regarding other implementation frameworks of message level security.

9 REFERENCES

[1]http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf

[2]K. Bhargavan, C. Fournet, A. Gordon, and G. O'Shea An Advisor for Web Services Security Policies, http:// research.microsoft.com/~adg/Publications/details.htm#sws05

[3]Microsoft Research; http://research.microsoft.com/projects/Samoa/

[4]K. Bhargavan, C. Fournet, and A. D. Gordon. Verifying policy-based security for web services. In *11th ACM Conference on Computer and Communications Security (CCS'04)*, pages 268–277, October 2004.

[5]T. Nadalin, ed., *Web Services Security Policy Language (WS-SecurityPolicy)*,Version 1.0, 18 December 2002, http://www.verisign.com/wss/WSSecurityPolicy.pdf

[6]K. Bhargavan, C. Fournet, A. D. Gordon, and R. Pucella. TulaFale: A security tool for web services. In *International Symposium on Formal Methods for Components and Objects (FMCO'03)*, LNCS. Springer, 2004

[7]B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, pages 82–96. IEEE Computer Society Press, 2001.

[8]Ana C.C. Machado and Carlos A. G. Ferraz. Guidelines for Performance Evaluation of Web Services, *WebMedia'05*, December 5-7,2005

[9]Hongbin Liu, Shrideep Pallickara Geoffrey Fox, Performance of Web Services Security

[10]XML-Signature Syntax and Processing http://www.w3.org/TR/xmldsig-core/

[11]Bajaj, et al., *Web Services Policy Framework (WS-Policy)*, September 2004,http://www.ibm.com/developerworks/library/specification/ws-polfram/

[12]T. Nadalin,ed.,*WS-PolicyAssertions*, 28 May 2003,http://www.ibm.com/developerworks/library/ws-polas

[13] Java Web Services Tutorial http://java.sun.com/webservices/docs/2.0/tutorial/doc/index.html

[14]Stefan Batres, ed., *Web Services Reliable Messaging Policy Assertion (WS-RM Policy)*, February 2005, http://specs.xmlsoap.org/ws/2005/02/rm/WSRMPolicy.pdf

[15]G. Della-Libera, M. Gudgin, P. Hallam-Baker, M. Hondo, H. Granqvist, C. Kaler, H. Maruyama, M. McIntosh, A. Nadalin, N. Nagaratnam, R. Philpott, H. Prafullchandra, J. Shewchuk, D. Walter, and R. Zolfonoon. Web services security policy language (WS-SecurityPolicy), July 2005. Version 1.1.

[16]Web Services Addressing (WS-Addressing) W3C Member Submission 10 August 2004http://www.w3.org/Submission/ws-addressing/

[17]SOAP, http://www.w3.org/TR/soap/

[18]http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnglobspec/html/ws-routing.asp

[19]Roubtsov,V. My kingdom for a good timer, http://www.javaworld.com/javaworld/javaqa/2003-01/01-qa-0110-timing.html

[20]http://specs.xmlsoap.org/ws/2005/02/sc/WSSecureConversation.pdf

[21]Microsoft Corporation. *Web Services Enhancements (WSE)2.0 SP1*, July 2004.At http://msdn.microsoft.com/webservices/building/wse/default.

[22]http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wse/html/40c4b84a-a6e8-40db-810e-2521fdd8c09d.as

[23]K. Iwasa, et al., eds., WS-Reliability v1.1, OASIS Web Service Reliable Messaging TC, OASIS Standard, 15 November 2004, http://www.oasisopen.org/committees/download.php/9330/WS-Reliability-CD1.086.zip.