Journal of Al Azhar University-Gaza (Natural Sciences), 2011, 13 : 1-16

Detection of XML Rewriting Attack: Enhance Inline Approach by Element Position

Tawfiq S. Barhoom
tbarhoom@iugaza.edu.ps
Raed S. K. Rasheed
rrasheed@iugaza.edu.ps
Islamic University-Gaza
Received 21/2/2010 Accepted 13/2/2011

Abstract: *Web services communicates with each other using an XML-based message, called Simple Object Access Protocol SOAP message which has all XML document characteristics including XML security.*

The SOAP message has much vulnerability. One of the vulnerabilities is modifying the SOAP message using unauthorized access. This case is called XML rewriting attack. Detecting the XML rewriting to ensure the security of SOAP requested by inline approach has some limitations and weaknesses. In this paper, we propose an enhancement of the inline approach through the element position of SOAP message elements using a tree-like structure, and implementation were presented

Keywords*: Element Position, Inline Approach, XML Rewriting Attack, SOAP message..*

I. INTRODUCTION

In spite of, the existence of web service specifications WS* (ex. WS-Security, WS-Policy and etc.), still there are many vulnerable cases with web service at message level if incorrect usage of this standard is caused by a human. Regarding the SOAP message used to communicate the web services together, we observe that SOAP message as an XML document having all the characteristics of the XML document. So, many vulnerable cases we can find in XML's document. One of these cases the XML rewriting which means injecting the XML document with new elements to modify the

document. This technique is used to attack the SOAP message maliciously using unauthorized access identity theft, etc [8]. The authors of [9] introduce an inline approach for protecting the integrity of the SOAP message using a structure called SOAPAccount. Later, they discuss the forging of the SOAPAccount itself [4] presenting a solution by using Check SOAPAccount module. In [8], the inline approach is explained and its limitations are demonstrated. Additional analyze in [11] shows the inline approach weakness. We propose in this paper how we can enhance the SOAPAccount structure to reduce vulnerable cases and to detect any XML rewriting attack and implementation were presented.

1. Definitions

1.1. **XML**

Extensible Markup Language (XML) is inspired by the Standard Generalized Markup Language SGML. XML concerns the data structure and describes the structure of data, not the format of data. It is presented as a class of data objects called XML documents, and partially describes the behavior of computer programs which process them [1].

1.2. SOAP Message

Simple Object Access Protocol (SOAP) is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment. It uses XML technologies to define an extensible messaging framework, providing a message construct that can be exchanged over a variety of underlying protocols. The framework has been designed to be independent of any particular programming model and other implementation specific semantics [2].

A SOAP message is encoded as an XML document, consisting of an Envelope, Header, Body and Fault elements. The <Envelope> is the root element in every SOAP message, and containing two child elements, an optional <Header> and a mandatory <Body>. The SOAP <Header> is an optional sub-element of the SOAP envelope, that can be used to pass application-related information that is to be processed by SOAP nodes along the message path. The SOAP <Body> is a mandatory sub-element of the SOAP envelope, which

contains information intended for the ultimate recipient of the message. The SOAP <Fault> is a sub-element of the SOAP body, used for reporting errors. With the exception of the <Fault> element, which is contained in the <Body> of a SOAP message [3], Figure-1 shows the main elements of a SOAP message. Figure-2 shows sample of a SOAP message document.

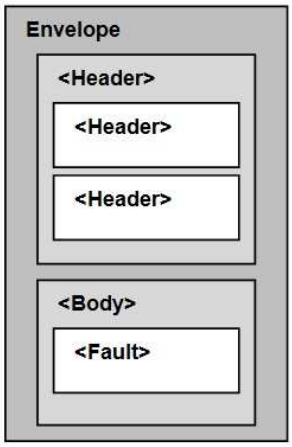

Figure 1: The elements of a SOAP message

```
<Envelope>
   <Header>
      <Header01>
         ....
      </Header01>
      <Header02>
         ....
      </Header02>
   </Header>
   <Body>
      ....
      ....
      <Fault>
         ....
      </Fault>
   </Body>
</Envelope>
```

Figure 2: Sample of a SOAP message

1.3 XML Rewriting and Attack - *XRaA*

SOAP message is an XML-based document. One particular vulnerable case is that of a XML rewriting attack – XML wrapping attack – which is a general name for a distinct type of attacks based on the malicious interception, manipulation, and transmission of SOAP messages in a network of communication system. Figure-3 shows SOAP message before attacking, using security and signed elements. Figure-4 shows how the attacker rewriting SOAP message functions by copying the old body and passing it into a new header. The signed element of Id 1 is still valid. Using WS-Security [5], WS-Policy [6] and other standards correctly on SOAP we can avoid XML rewriting attacks [7]. However, in practice, incorrect deployment of these standards specially by human being, is very likely, leading to significant vulnerable cases [4].

II. Related work

There are many mechanisms that have been previously proposed in order to secure WS communications. SOAP Account [4] [9], WS-Policy [10], WSE Policy Advisor [7] and Formal methods can be used [8], but these proposals are not sufficient to detect all types of XML rewriting attack.

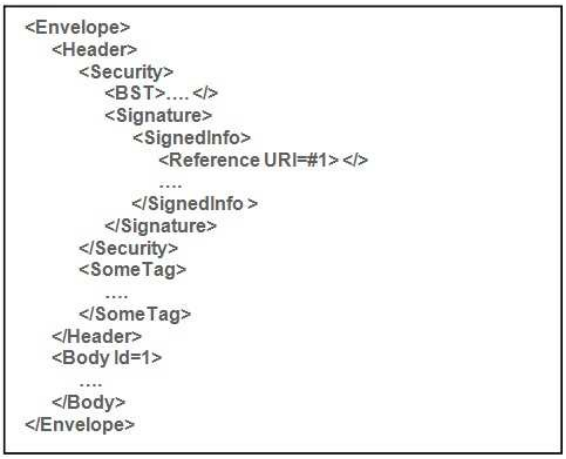

Figure 3: SOAP message before XML rewriting

Figure 4: SOAP message after XML rewriting

Benameur, Abdul Kadir, and Fenet XML [8] discuss the XML

rewriting attack and list the existing solutions such as SOAPAccount, WS-Policy, WSE Policy Advisor and the formal methods, explaining their limitations. Gajek, Liao, and Schwenk [11], proposed fixing the inline approach by retuning XPath with position information.

III. Inline approach and detected XRAA

3.1. Inline Approach and SOAP Account

SOAP Account [9] is a structure information used as a header in the headers. This header is used to detect the XML rewriting attack, and protect the SOAP message integrity; its structure consists of the following elements:

- The number Of Child Elements of Envelope.
- The number Of Header Elements in the SOAP message.
- The number Of References in each signature.
- Successor and Predecessor Relationship of Each Signed Object: Parent Element and Sibling Elements.
- A Possible extension for future improvement.

Back to the SOAP message in Figure-3 we can introduce the SOAPAccount as:
The number Of Child Elements of Envelope = 2.
The number Of Header Elements = 2.
The number Of References in each signature =1.
Parent of Element “Id 1” is Envelope.
Sibling of Elements “Id 1” is Header only.

After performing the Add SOAP Account module [9], the SOAP message will be added by the SOAP Account as a new header for validating the integrity of the SOAP message. Figure-5 shows the SOAP message after adding the SOAP Account. When the SOAP message is attacked as in Figure-4 the SOAP Account will detect the modification and protect the SOAP message integrity.

Figure-6 shows SOAPAccount with the attacked SOAP message, and how can the attack be detected. The determination can be performed by comparing the NoOfChildOfHeader in SOAPAccount element before attack (was 3) and after attack (is 4). According to [4] [9] the CheckSOAPAccount module checks' the validity of SOAPAccount. If any attacking SOAPAccount itself and the

validation of SOAPAccount integrity have been done before the validation of SOAP message, this means that the SOAPAccount is not valid, and the SOAP message is already attacked.

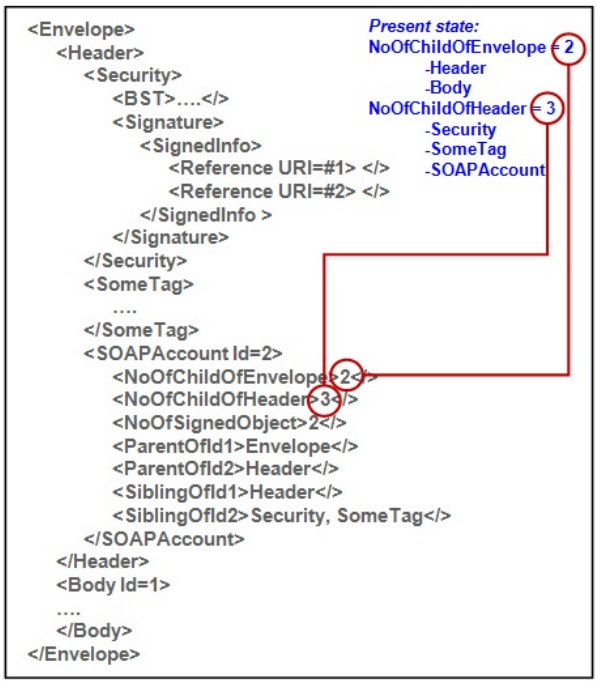

Figure 5: SOAP message after adding SOAP Account

3.2. Element Relationship Weakness

One of the weaknesses of the inline approach is that the SOAPAccount only preserves the relationship to its parent and siblings elements. This is a relative position in the DOM tree. To mitigate the vulnerability, one recommendation [11] is to consider in SOAPAccounts the absolute path to the root element ("vertical fixing") and to its siblings ("horizontal fixing"). Figure-7 shows the weakness and how an attacker can modify the SOAP message by copying the envelope into another header and the SOAPAccount cannot detect this attack.

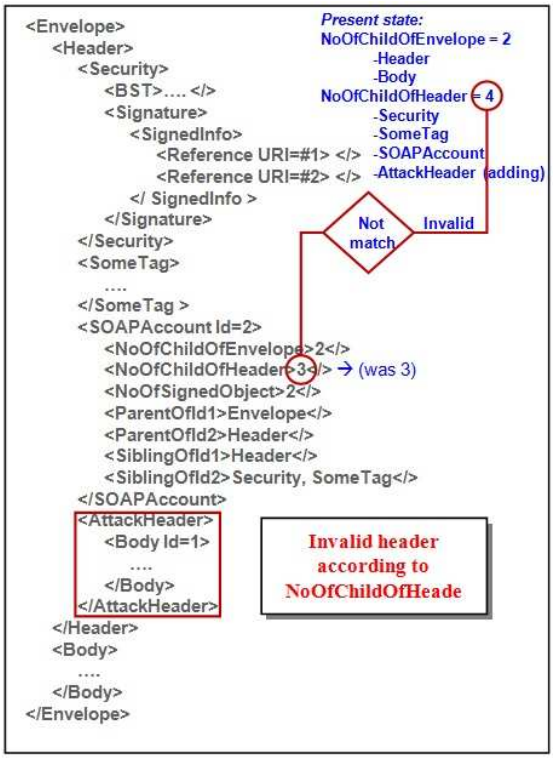

Figure 6: SOAP Account detect the attack

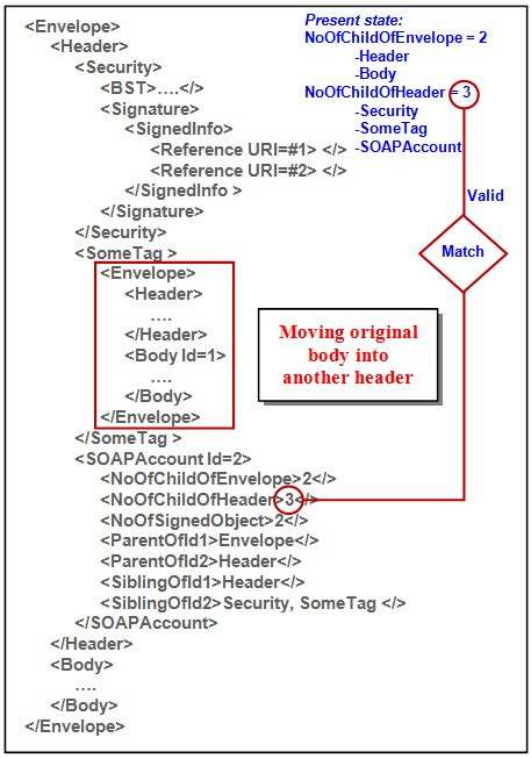

Figure 7: SOAP Account cannot detect the attack

IV. Inline approach enhancement

SOAP message is nothing but XML document, that we can represent by using a tree structure, Figure-8 shows the SOAP message represented as tree and its SOAPAccount will be as follows:
The number Of Child Elements of Envelope = 2.
The number Of Header Elements = 3.
The number Of References = 2.
Parent of Element “Id 1” is Envelope.
Parent of Element “Id 2” is Header.
Sibling of Elements “Id 1” is Header only.
Siblings of Elements “Id 2” are Security and SomeTag.

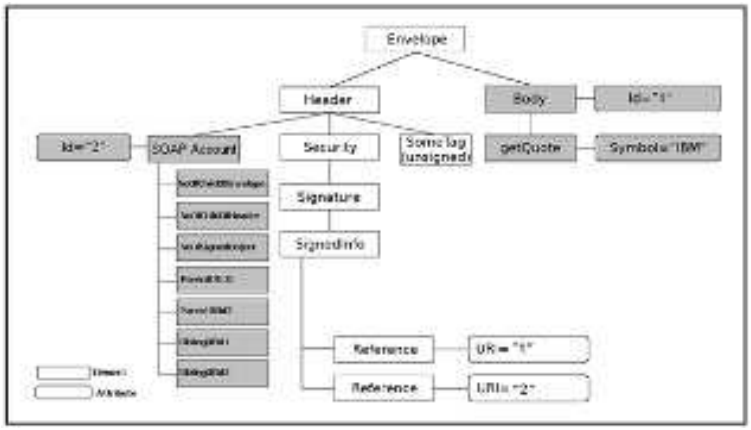

Figure 8: SOAP message tree

Figure-9 shows the SOAP message weakness, after the attack, the SOAPAccount is still valid and cannot detect the attack. So to detect attacks like this, look at SOAP message as a structure tree, therefore we will observe that we can determine the position of all tree nodes easily and early after creating SOAP message directly. For example, attacker changes the body from quote symbol IBM to new quote symbol MBI which means the ultimate receiver will perform the new body, and ignore the original one.

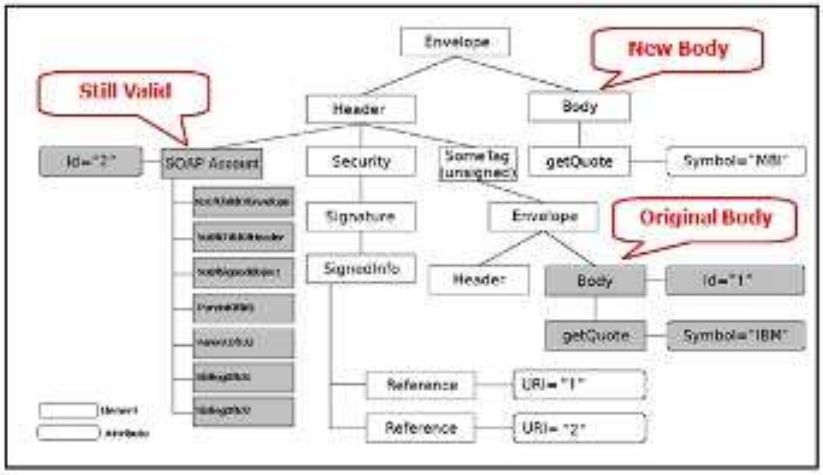

Figure 9: SOAP message tree after attack

Using a post-order traverse by visiting firstly left node, then right

node, and finally root node, we can determine the element position and keep it as shown in Figure-10 and Figure-11 as tree. According to the importance of the signed elements we will store the element position in an attribute in the predecessor element and call it elementPosition attribute. The SOAPAccount for Figure-10 will be:
The number Of Child Elements of Envelope = 2.
The number Of Header Elements = 3.
The number Of References = 2.
Parent of Element “Id 1” is Envelope and its position is 18.
Parent of Element “Id 2” is Header and its position is 15.
Sibling of Elements “Id 1” is Header only.
Siblings of Elements “Id 2” are Security and SomeTag.

If we perform the validation of SOAPAccount with new elementPosition attribute invalid position of <Header> will be detected immediately. Figure-12 shows the element position after attack.

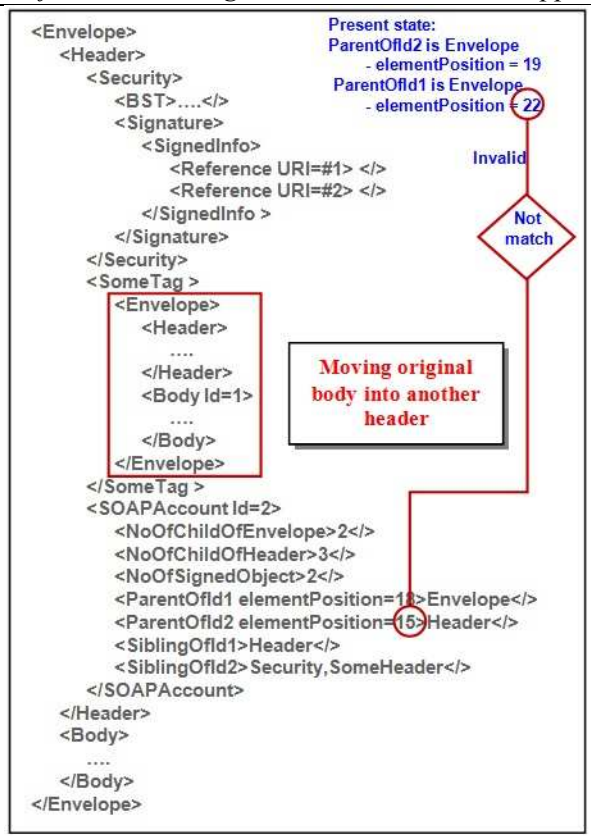

Figure 10: elementPosition attribute detect the XML rewriting

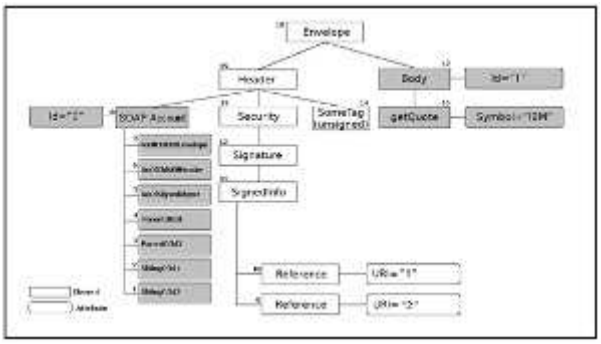

Figure 11: SOAP message tree with element position

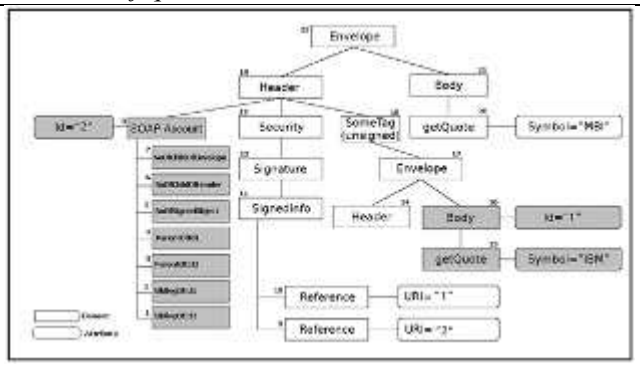

Figure 12: SOAP message tree with element position after attack

V. Empirical Work

The main idea of enhancement the inline approach is to add the elementPosition attribute to the SOAPAccount structure at the <ParentOfId> element which contain the parent of the signed element, so we found the function *setElementPosition* which add the elementPosition attribute into the <ParentOfId> element to be added to the *AddSOAPAccount* model proposed by [12] and the function *checkElementPosition* which check the validation of the elementPosition and add it to the *CheckSoapAccount* model proposed by [12] which presented at Figure-13 with founded functions Figure-13 shows where the developed functions *setElementPosition* and *checkElementPosition* are imbedded into the overall model proposed by [12], the function's code shown in Figure-14 and Figure-15.

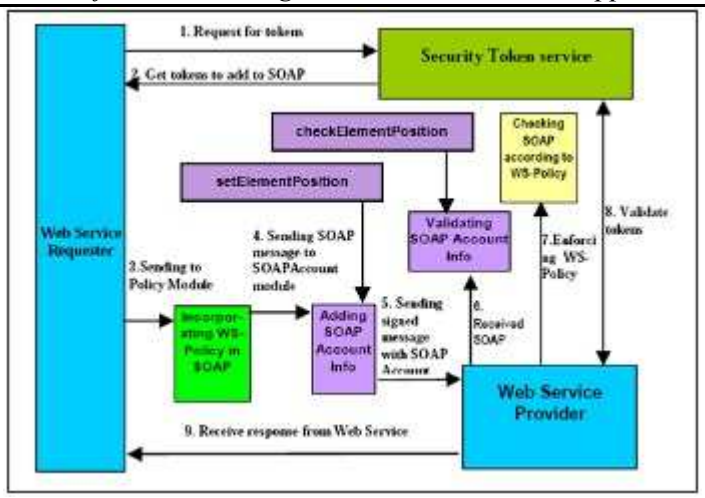

Figure 13: Message flow using inline approach and proposed setElementPosition and checkElementPosition functions

After applying the proposed method on several SOAP messages and produce SOAP messages with elementPosition attribute, the method used to detect the attack which could not be detect using the inline approach (the attack has been mentioned in section IV). The attacked SOAP message at Figure-16 and the result of *checkElementPosition* function is presented in Figure-17. The function *checkElementPosition* detect the attack and present an error which is wrong element.

```
public static void setElementPosition(Node ele){
    for(Node n = ele.getFirstChild(); n!=null ; n=n.getNextSibling() ){
        if(n.getNodeName().startsWith("ParentOfId")){
            Node n1=n.getFirstChild();
            String s = findParentePosOfSignedElement(n1.getNodeValue());
            Element e = (Element) n;
            e.setAttribute("elementPosition",s);
        }
        setElementPosition(n);
    }
}
```

Figure 14: setElementPosition function.

```
public static void checkElementPosition(Node ele){
    for(Node n = ele.getFirstChild(); n!=null ; n=n.getNextSibling() ){
        if(n.getNodeName().startsWith("ParentOfId")){
            Node nl=n.getFirstChild();
            String s = findParentePosOfSignedElement(nl.getNodeValue());
            Element e = (Element) n;
            if(!e.getAttribute("elementPosition").equalsIgnoreCase(s)){
               System.out.println("Error: e"+nl.getNodeValue()+
                                  "'s position is "+
                                  e.getAttribute("elementPosition")+
                                  " not "+s);
            ePosError = true;
            }
        }
        checkElementPosition(n);
    }
```

Figure 15: checkElementPosition function.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Header>
    <Security>
      <BST>
      </BST>
      <Signature>
        <SignedInfo>
          <Reference URI="1"><DigestValue>Ego0kjha245vcas</DigestValue>
          </Reference>
          <Reference URI="2"><DigestValue>somaspaa2mazvkelw</DigestValue>
          </Reference>
          <Reference URI="3"><DigestValue>somaspaa2mazvkelw</DigestValue>
          </Reference>
        </SignedInfo>
      </Signature>
    </Security>
    <SomeTag Id="3">
      <soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Header>
    <Security>
      <BST>
      </BST>
      <Signature>
        <SignedInfo>
          <Reference URI="1"><DigestValue>Ego0kjha245vcas</DigestValue>
          </Reference>
          <Reference URI="2"><DigestValue>somaspaa2mazvkelw</DigestValue>
          </Reference>
          <Reference URI="3"><DigestValue>somaspaa2mazvkelw</DigestValue>
          </Reference>
        </SignedInfo>
      </Signature>
    </Security>
  </soap:Header>
  <soap:Body Id="1">
    <m:GetStockPrice xmlns:m="http://www.example.org/stock">
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
    </SomeTag>
      <SOAPAccount Id="2">
        <NoOfChildOfEnvelope>2</NoOfChildOfEnvelope>
        <NoOfChildOfHeader>3</NoOfChildOfHeader>
        <NoOfSignedObject>2</NoOfSignedObject>
        <ParentOfId1 elementPosition="25">soap:Envelope</ParentOfId1>
        <ParentOfId2 elementPosition="21">soap:Header</ParentOfId2>
        <ParentOfId3 elementPosition="21">soap:Header</ParentOfId3>
        <SiblingOfId1>soap:Header</SiblingOfId1>
        <SiblingOfId2>Security,SomeTag</SiblingOfId2>
        <SiblingOfId3>SomeTag,SOAPAccount</SiblingOfId3>
      </SOAPAccount>
  </soap:Header>
  <soap:Body>
    <m:GetStockPrice xmlns:m="http://www.example.org/stock">
      <m:StockName>MBI</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
```

Figure 16: The attacked SOAP Message.

position related to the current node position. For example the first line of Figure-17 shows that the current position of <soap:Envelope> element is 40 but the actual position is 25 which mean that there is an attack.

```
Erro: <soap:Envelope> position is 25 not 40
Erro: <soap:Header> position is 21 not 36
Erro: <soap:Header> position is 21 not 36
Checking elementPosition complete, an attack has been found..
```

Figure 17: The result after checkElementPosition check the attacked SOAP Message.

VI. Conclusion

The rewriting attacks are discussed, and the inline approach to solve it, is explained. The proposed solution to enhance the inline approach is adding an attribute to the element point to the signed element. This attribute is called elementPosition which can detect any modification of SOAP message using the XML rewriting attacks. To detect the attack there are two main functions implemented *setElementPosition and checkElementPosition.*

REFERENCES

[1] Extensible Markup Language (XML) 1.1 (Second Edition) W3C Recommendation 16 August 2006, edited in place, 29 September, 2006 http://www.w3.org/TR/xml11

[2] SOAP Version 1.2 Part 1: Messaging Framework (Second Edition) W3C Recommendation 27, April, 2007 http://www.w3.org/TR/soap12-part1

[3] http://www.ibm.com/us/en/

[4] M. A. Rahaman, A. Schaad, and M. Rits, Towards Secure Soap Message Exchange in a soa. In Workshop on Secure Web Services, 2006.

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

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

[7] K. Bhargavan, C. Fournet, A. Gordon, and G. O'Shea, 2005. An Advisor for Web Services Security Policies.

[8] A. Benameur, F. Abdul Kadir, and S. Fenet. XML Rewriting Attacks: Existing Solutions and their Limitations. In IADIS Applied Computing 2008. IADIS Press, Apr. 2008.

[9] M. A. Rahaman, R. Marten, and A. Schaad. An inline approach for secure soap requests and early validation. OWASP AppSec Europe, 2006.

[10] Web Service Security Policy, 2005. http://specs.xmlsoap.org/ws/2005/07/securitypolicy/wssecuritypolicy.pdf

[11] S. Gajek, L. Liao, and J. Schwenk. Breaking and fixing the inline approach. In SWS'07: Proceedings of the 2007 ACM workshop on Secure web services, pages 37–43, New York, NY, USA, 2007. ACM.

[12] M. A. Rahaman, A. Schaad, and M. Rits. Towards secure soap message exchange in a soa. In Workshop on Secure Web Services, 2006.