CHAPTER 5

SELF-AWARE MESSAGE VALIDATING ALGORITHM FOR PREVENTING XML BASED INJECTION ATTACKS

5.1 INTRODUCTION

Web Service attacks, generally called XML-based attacks, occur at the SOAP message level and thus they are not readily handled by existing security mechanisms in earlier firewalls. So as to provide robust security mechanisms for Web Services, XML filters have recently been introduced for Web Services security. In this research, a framework for dynamic XML filters are proposed, called self-aware message validating filter for XML based attacks, which supports detection and protection of XML-based attacks in real-time. A detailed design of the injection filter security model has been provided by validating schema information of the message with detection and protection policies. The information in the form of SOAP request is passed between client and server. Then it is processed in the Web Services opening an array of XML based injection attacks, for example, oversized message, message replay, parameter tampering, coercive parsing and semantic URL attacks. These attacks, among others, will be the focus of this section.

The solution to protect a Web Services system from the XML-based injection attacks are to understand its threat outline and to study how threats may impact the performance of the Web Services system. Many organizations such as Microsoft and OWASP tried to mitigate major class of threats to Web Services for the purpose of protecting Web Services systems
more effectively. Attacks to Web Services are normally XML-based attacks that can be raised on Web Services via messages, which depend on XML as its message format and application layer protocols. Typical XML-based attacks include XSS injection attack, XPath injection attack, oversized message attack, replay attack, parameter tampering attack, XML injection attack, SQL injection attack and coercive parsing attack. For example, an XSS injection attack takes advantage of the weakness of CDATA of the parser of a service provider to allow malicious script in XML documents, forms or other methods in order to deform the information of the Website.

An oversized message attack is a type of flooding attacks, where an attacker creates enormous level of traffic to a Web Service to exhaust its resources at the server side and parameter tampering attack can crash the server by sending unacceptable parameters. An XML-based attack can also be in a form of a distributed multi-faceted attack, Most of the XML-based attacks come in unpredicted format and so far the performance has not been studied carefully. For example, the attacker finds the XML tag for administrator then inserts that tag to act as high privileged user like administrator. This privilege violation would not be prevented the conventional firewall. Thereafter the attacker imitates the administrator’s activities and gains all details of user realms.

Many security approaches have been developed for protecting Web Services, but they are vulnerable to predict and prevent the variety of attacks such as an XDoS attack. Some business concerns hesitate to adopt service oriented technologies because of lacking technology of robust security mechanism to prevent XML based attacks. An approach to defend against XML-based attacks at the application level is achieved in this work through self-aware message validating algorithm is proposed. It supports detection of XML-based attacks in real-time.
In this research, the validation approach is used as a security mechanism and presented a framework for XML based injection filter.

5.2 ARCHITECTURE OF XML BASED INJECTION FILTER SERVICE

![Figure 5.1 A framework for injection filter service](image)

The architecture of the XML based injection filter service model is illustrated in Figure 5.1. As shown in the figure, an injection filter lies between service consumers and a service provider, and can be installed either on the same or a different machine where the actual Web Services are
deployed. It interacts with service consumers through its User Interface (UI), which is responsible for receiving requests from and sending responses back to the login service.

There are five major components supporting the filters, namely oversized message filter, message replay filter, parameter tampering filter, coercive parsing filter and semantic URL filter, which process the incoming request and state-based information, respectively. In the injection filter security model, input validation and protection are the major features for providing user access control, which ensure that only valid users are allowed to access certain Web Services.

5.2.1 Parameter Tampering Filter

The attacker adjusts the parameters in a SOAP message in an attempt to redirect the input validation in order to access unauthorized information. A change in integrity of the parameters is to detour the input validation and gains unauthorized access of some confidential functionality of Web Services. Because, the input parameters of an operation are given within a WSDL document, the hacker can play with different combinations of parameter patterns in order to access the unauthorized information. This filter checks the XML schema definition of received message for data type, null values. This filter checks the parameter for valid data. If it fails, then, it throws an error to the sender once. Even if the sender continues, his/her misbehaving with parameters leads to the disconnection of communication.

5.2.2 Coercive Parsing Filter

This filter verifies the namespaces and version mismatch received in the WSDL and SOAP files. This filtering policy used the fault values in
SOAP fault code. The filter verifies the received message for wrong format of SOAP message by generating SOAP fault code. This filter blocks the input that has a strange format. This policy used the values in SOAP fault code. They are version mismatch and must understand fault code.

5.2.3 Oversized Message Filter

The XML parsing of the service provider is directly affected by the size of the SOAP message. As a consequence, large amounts of Central Processing Unit (CPU) cycles are consumed when presented with large documents to process. A hacker can send a payload that is in alarming rate to exhaust systems resources. So, the filter is designed to refine the size of the message, requisition resources presented in the incoming message. Hence, the filter does the checking of three important parameters for the received SOAP message. First, it sets a request timeout to prevent infinite delay attacks. Then, it limits the amount of data that it retrieves.

5.2.4 Message Replay Filter

A hacker can resend the SOAP message requests to access the Web Service using other’s login credentials. This kind of Web hacking will be escaped as a legitimate request because the source IP address is valid, the network packet attributes are valid and the HTTP request is well formed. Though, the business behavior of replay attacker is valid with unmatched parameters of the information is treated as an XML intrusion. Hence, the filter assigns an identifier for each incoming message and stored into the database to identify the replayed message. After that, the filter catches and matches the identifier of incoming messages and uses the replay detection policy to identify and reject messages which match an entry in the database of replay detection filter.
5.2.5 Semantic URL Filter

The semantic URL attack is the client manually retypes the parameters of its request by keeping the URL’s structure but altering its semantic meaning. This is protected by giving token and timestamp for expiration. In this way, the filter is designed to protect the Web Service provider from the attack. The server and client shares the NONCE (Number used ONCE) for mutual identification. The random number generator is used the process identity of task to generate the nonce. Then, it is added with time stamp calculator to maintain the message expiration of the client/server. These parameters are shared either of the client/server to protect the integrity of the message.

5.3 ALGORITHM OF SELF-AWARE MESSAGE VALIDATION FOR XML BASED ATTACK INJECTION FILTER

A Web Service communicates with other applications over a network. There is no surety that the incoming message is requested from legitimate user, though the incoming request coming from authorized IP address. Meanwhile, the intruder includes some parameter to gain some data or to redirect the flow to some proprietary Web links. Based on the input information, the XML Request Handler can detect and verify XML based attack in real-time. The corresponding tables are created in the user information databases, which are used to store not only the current state and user information, but also the previous states and recent user information that are useful for attack detection and verification.

5.3.1 Detection of XML Based Injection Attacks
The XML request handler module is responsible for the dynamic detection and verification of the XML-based injection attacks by checking both the SOAP message and the parameters passed to a Web Service operation. The algorithm proposed by the XML request handler module is depicted in Figure 5.2 and explained how the input is treated as malformed input or not. As shown in the figure, when a SOAP message with a valid user request is sent to the XML request handler module, there the input is refined in all filters to verify the attack. This has been implemented by SAX parser of the server side filter to receive the legitimate schema of incoming message.

<table>
<thead>
<tr>
<th>Algorithm for XML Based Attack Injection Filter Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input: SOAP message with parameters for service invocation</td>
</tr>
<tr>
<td>Output: accept / reject</td>
</tr>
</tbody>
</table>

1. Receive SOAP message
2. switch (input)
3. case Parameter tampering attack:
4. verify parameter tampering attack using tamper filtering polices
5. case Coercive parsing attack:
6. verify coercive parsing attack using parsing filtering polices
7. case Oversized message attack:
8. verify oversized message attack using oversized message filtering polices
9. case Message replay attack:
10. verify message replay attack using replay filtering polices
11. case URL semantic attack:
12. verify URL semantic attack using URL semantic filtering policies
13. if any attack confirmed
14. then forward incoming request to XML request handler
15. output reject the attacker’s request
16. else output is valid and invoke a login service

Figure 5.2 Algorithm for XML based injection filter

The process of detecting other types of XML-based attacks involves two major steps, which are detection of malformed SOAP messages and protection from attacks. Malformed SOAP messages are detected using the SOAP message validator. For example, to detect XML attacks, the handler module analyzes for possible flooding requests and keeps track of the allowable message size and the nesting depth in the incoming XML messages. If a certain type of attack is detected, the handler module will attempt to
verify the attack using additional evidence from the blacklist database. Once an attack is confirmed, the SOAP message is rejected, and sent to the XML request handler module, where a rejection message is generated and sent back to the service consumer. In addition, the blacklist database is updated accordingly with the information related to the attack. Otherwise, the SOAP message is sent to the deployed Web Service for service invocation, and after the service invocation, the results are sent back to the service consumer.

The XML based attacks are categorized into five significant attacks and the protection mechanisms are implemented to refine and report the malformed input. First, the input filter is received in server side is passed to all filters to find the attack vectors. There, the input is passed to various input verification policies on parsers. Next, the captured input containing attacks are blocked and reported in XML request handler. Last, the legitimate input is forwarded to service provider.

5.3.1.1 Parameter tampering filter

In this, filter the received parameters are checked for data type, number of parameter and null values. This filter checks the parameter for valid data; if it fails, then, it throws an error to the sender once.

<table>
<thead>
<tr>
<th>Algorithm for parameter tampering attack filter</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> request message</td>
</tr>
<tr>
<td><strong>Output:</strong> accept/reject</td>
</tr>
<tr>
<td>1. Call tampercheckerXml(reader)</td>
</tr>
<tr>
<td>2. If (reader == null or empty)</td>
</tr>
<tr>
<td>send ArgumentNullException(&quot;reader&quot;)</td>
</tr>
<tr>
<td>3. Get xsdpath from received XML document</td>
</tr>
<tr>
<td>4. GetAttribute(&quot;xsdPath&quot; as xsdpath)</td>
</tr>
<tr>
<td>5. If (xsdpath != null or xsdpath!=empty){</td>
</tr>
<tr>
<td>assign xsdPath = xsdpath;</td>
</tr>
<tr>
<td>else send Exception for MissingXsdPath;</td>
</tr>
<tr>
<td>6. Get StartElement from received XML document</td>
</tr>
<tr>
<td>7. If(StartElement != Empty)</td>
</tr>
<tr>
<td>ReadEndElement</td>
</tr>
<tr>
<td>8. If(EndElement != Empty)</td>
</tr>
</tbody>
</table>
Call Web method

else

send exception to the attacker and alert administrator

---

**Figure 5.3 Algorithm for parameter tampering attack filter**

Even if the sender continues, his misbehaving with parameters leads to the disconnection of communication. To solve this problem, the proposed XSS filter was created with tamperchecker function as given in Figure 5.3. It checks the arguments for null values, data type, start element and end element of the received request from the client.

The validation process itself is hidden from the client. The message validation of parameter tampering filter makes a number of checks to validate the message. That includes the verifying of the message payload is well-formed, means to verify whether the document follows all rules of recommended by W3C or not, also ensures to a predefined schema with acceptable data types and range of values. If the request succeeds all the validation checks that are performed by the message validator, then the service processes the message and forwards the request to the Web Service provider.

**Advantages**

This filter protects the server from anomaly parameters and unstructured form of XML document. This assists to guard against the injection attacks, even for business Web Services that do not implement any validation control. The Web Service performs validation autonomously for the client.

**5.3.1.2 Coercive parsing filter**
This attack is targeted on the standard structure of XML documents. In Web Services, every document must be formatted according to the rules and regulations mentioned in the W3C consortium. If any misplaced namespaces, any disparity in version of SOAP or WSDL document will affect the communication greatly is shown in Figure 5.4. This malformed attack stops the service provider suddenly, after affected by enormous level of attacks.

- **Version Mismatch Fault Code**

  It finds an invalid namespace for a SOAP envelope and throws exception.

- **Must Understand Fault Code**

  It indicates whether a header entry is compulsory or optional for the recipient to process.

---

**Algorithm for coercive parsing filter**

<table>
<thead>
<tr>
<th>Input: XML document</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: accept / reject</td>
</tr>
</tbody>
</table>

1. Get XML document
2. If (SOAP version==valid namespace)
   - if(mustUnderstand == 1)
     - allow
   - else throw mustunderstand Fault Exception
3. else throw Version Mismatch Fault Exception

**Figure 5.4 Algorithm for coercive parsing filter**
5.3.1.3 Oversized message filter

Denial of Service attacks happened by exhausting resources available in server side. Such attacks aim at reducing service availability by exhausting the resources of the service’s host system, like memory, processing resources or network bandwidth. It is performed through query a service using a very large request message, which is called as over sized message.

An oversized message attack is simple to perform, due to the high memory consumption of XML tags and its long processing duration. The total memory utilization to process one SOAP message is higher than the message size.

In-built security standards like WS-Security only concentrate on message integrity, confidentiality, authentication and authorization. In this research, a system for protecting Web Services from oversized message attacks is presented. Over sized message attacks often exists on malformed or extremely lengthy messages that makes the server busy in resource-consuming computations. Therefore, provider needs a suitable mitigation mechanism to prevent such kinds of attacks is the validation of messages by self-aware message validating filter before forwarding them to the server. We discussed over sized message attacks against Web Services, show how the XML schema can automatically be derived from formal Web Service descriptions (written in the WSDL), and presented an oversized message filter to protect Web Service messages.

Because of oversized message, the Web Service resources such as CPU time, memory usage and database connections keep busy. The filter was
implemented to measure the size of the incoming message. Moreover, it verifies the signature to ensure that the message is not transferred in transit. Finally, it parses the entire request message for malicious content. In this way it is designed to find buffer overflow attack.

![Diagram](image)

**Figure 5.5 Over sized message validation of filter from server side**

In the filter, the incoming message is checked for three important parameters. First, it sets a request timeout to prevent infinite delay attacks. Then, it limits the amount of data that it will retrieve. Last, it restricts the message from retrieving resources on the local host as shown in Figure 5.5.

The algorithm is shown in Figure 5.6 depicts how it sets the values of attributes. The service provider had to set its maximum request length. The filter loaded the XML document and change XSD value maxMessageLength=1024 or any required buffer size. This compares the size of the request against the maximum allowable size that is specified for request messages.
The algorithm is given below:

**Algorithm of oversized message filter**

Input: XML document  
Output: accept / reject  

1. Get XML document of client  
2. Get messaging attribute  
3. it sets a request timeout to prevent infinite delay attacks  
4. If maxMessagingLength == 1024 then do Web method  
5. Else if Responsevalue>TimeOut  
   reject and send the alert message  
6. else accept  

---

**Figure 5.6 Algorithm of oversized message filter**

In the filter, the incoming message is checked for three important parameters. First, it sets a request timeout to prevent infinite delay attacks. Then, it limits the amount of data that it will retrieve. Last, it restricts the message from retrieving resources on the local host.

**5.3.1.4 Message replay filter**

An attacker attempt to resend SOAP requests to repeat sensitive transactions is called the message replay attack. Here, the client side message is assigned with an identifier and time stamp then send to the server for validation purpose. Thereafter, the filter captures the identifier of incoming messages and rejects messages that match an entry in the replay detection database. If the message identifier is valid because of its nonexistence, the filter compares the message timestamp to its clock time value for
synchronization. If the message identifier has unacceptable identifier or any time stamp mismatch then, the message is rejected. This can be done by calculating elapsedTime, cacheLifeSpan and maxMessagePeriod. The time tolerance is the acceptable value time difference between the sender and the maximum message period is configured as 600 seconds.

- The filter calculates the elapsed time period of the message by deducting the created time value on the message from the present server time.

- For a message that appears have been created in the past or if the server and message creation times are equal will be rejected.

- Otherwise the message will be accepted only when its message age is less than or equal to the values for the maximum message period parameter and the elapsed time setting.

- Filter checks the maximum message period value is less than or equal to the elapsed time setting to accept the message.

\[
\text{Cache Life Span} = (\text{MMP} + \text{ET} \times 2)
\]

<table>
<thead>
<tr>
<th>CLS</th>
<th>CacheLifeSpaninSeconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMP</td>
<td>MaximumMessagePeriod</td>
</tr>
<tr>
<td>ET</td>
<td>ElapsedTime</td>
</tr>
</tbody>
</table>

The algorithm of message replay filter is given in Figure 5.7. That accepts the XML document and retrieves its attribute through its parser. First it gets CLS and MMP attributes of incoming message. Then it calculates expiration to find that as an old message or new one. This is calculated by
subtracting current time and time stamp of the received message. Then the
difference value is assigned as the message period and stored into cache
database. Next, it checks for messages where sender's clock is slower than the
server’s clock for first condition. Finally, it accounts for messages where the
sender's clock is faster than the server’s clock through the second condition.
Then the identifier is stored into database.

The unique identifier for the message is collected and saved in the
replay finder cache before processing the request. The unique identifier is also
used to solve the concurrency message collision issues. In scenario of two
messages arrived at same time will be solved, if a second message arrives
before the first message has finished executing. Some of the steps to be
performed when attempting to detect replayed messages can harmfully affect
system response time. For example, verifying the identifier of each incoming
message and timestamp is computationally exhaustive will create XDoS.

<table>
<thead>
<tr>
<th>Algorithm of message replay filter</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> XML document</td>
</tr>
<tr>
<td><strong>Output:</strong> accept / reject</td>
</tr>
<tr>
<td>1. Create input/output filter for client/server.</td>
</tr>
<tr>
<td>2. Read XML document</td>
</tr>
<tr>
<td>3. Get CLS and MMP of incoming message.</td>
</tr>
<tr>
<td>4. Calculate the message expiration time based on the cache lifespan configured in the policy assertion.</td>
</tr>
<tr>
<td>5. Get the current time.</td>
</tr>
<tr>
<td>6. Compute the time difference between the message timestamp and the current time.</td>
</tr>
<tr>
<td>timeDifference = currentDate - timestamp;</td>
</tr>
<tr>
<td>7. Assign messagePeriod = timeDifference</td>
</tr>
<tr>
<td>8. Check for messages where sender's clock is slower than the server’s clock for first condition.</td>
</tr>
<tr>
<td>9. Account for messages where the sender's clock is faster than the server’s clock through the second condition.</td>
</tr>
<tr>
<td>if((messagePeriod&gt;maxMessagePeriod+ElapsedTimeInSeconds)</td>
</tr>
<tr>
<td>10. Add the Message, identity value of message and expiration time to message replay cache database.</td>
</tr>
</tbody>
</table>
The filter accesses or updates the message replay cache could burden the server and affects server’s response time. This is also another vulnerable point of the server. After analyzing the server’s vulnerable points, the intruder replays enormous amount of message to the server that leads the server keeps the message identifier in cache and calculates the time difference for flooded messages. If the server is busy with accessing the database for indefinite time period makes XDoS attack.

To alleviate this attack the replay attack filter maintains freshness period time for all messages. This is achieved by setting the clock skew value on the server to determine the allowable time period. If a message is reached beyond the allowable range, then the message will be rejected, even though it is not exist in the replay database. The clock skew is less than the time and that the messages are not present in the cache is accepted.

### 5.3.1.5 Semantic URL attack filter

This filter service acts to handle password reset request from the client. In addition it is implemented to handle semantic URL attack which is triggered by the legitimate user. The legitimate user can also attack the server to retrieve other’s password the URL link by changing its parameters when he received URL link for password reset. In this way the attacker tries to modify the password of the other user. This filter sets the nonce by generating a random number and assigns the time stamp to use the request URL for limited time. The random number generator needs a seed value to create the nonce. For that, the system takes the process id as seed value as given below.
NONCE = \((\text{PID})^C \mod N \| \text{TimeStamp}((\text{SD}+\text{ST})+D)\)

- PID - Process ID
- C - Counter value in 100 digits
- N - Large Prime number in 200 digits
- SD - System Date
- ST - System Time
- D - 24 Hours

The generated nonce is lengthy and tough to break by brute force attack, because PID is generated by a 32-bit memory operating system. Then the system increased its power with counter variable to create a more complex number. The URL concatenated with user identity and nonce.

Client URL=URL \| UID \| NONCE.

- URL - Uniform Resource Locator
- UID - User Identity
- NONCE - Number used ONCE

Algorithm for Password reset policy of the proposed semantic URL filter

<table>
<thead>
<tr>
<th>Input: reset request (user identity and password)</th>
<th>Output: accept/reject</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Client sends message M1=reset request | UID | PWD1 to the service provider</td>
<td></td>
</tr>
<tr>
<td>2. Service provider authenticates UID and calculates N=(PID)^C \mod N | TimeStamp((SD+ST)+D)</td>
<td></td>
</tr>
<tr>
<td>Concatenates Client URL=URL|UID|N|T_1</td>
<td></td>
</tr>
<tr>
<td>Sends M2=URL|UID|N|T_1 to the client</td>
<td></td>
</tr>
<tr>
<td>3. Calculates T2=T1+Time stamp of client</td>
<td></td>
</tr>
<tr>
<td>4. M3=UID|PWD1|PWD2|N|T2</td>
<td></td>
</tr>
<tr>
<td>5. Compares UID,N,PWD1 if (T2&lt;(T1+24hrs)) replaces PWD1=PWD2 then discard</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.8 Algorithm of semantic URL filter
The algorithm is proposed to send and receive the password resetting request securely is shown in Figure 5.8. It utilizes the dynamic nonce and used for each and every communication.

Therefore, the attacker cannot use the URL or imitate the user id embedded in the URL by substituting other user id and random numbers. The attacker cannot predict the seed value of nonce from the 32-bit operating system or higher configuration. Because each and every time the process id is changed by the operating system.

5.4 IMPLEMENTATION OF XML BASED INJECTION ATTACK FILTER

5.4.1 Parameter Tampering Filter

This filter prevents the service from processing messages that are not well-formed or that do not conform to an expected XML schema. The validation of input messages before deserializing them into native data types so that they can be interpreted as regular expressions.

The message validation validates each incoming request message to ensure that it is well-formed XML, that it contains all of the parts required by the service, and that the contents of the message conforms to an expected structure as defined by an XML Schema Definition (XSD). Then the regular expression checks to ensure that input contains only valid data and does not contain malicious SQL, HTML, or JavaScript code that could lead to code injection attacks. Afterwards, the validator ensures that complex data types (such as social security numbers and telephone numbers) are received in a format that the service can process. Figure 5.9 illustrates the process by which message validation logic intercepts request messages and verifies that they are acceptable for processing by the service.
The client sends a request message to the service. The service validates the message. The service uses a number of different validation checks to prevent malicious input. First, checking the format of the request message is to ensure that the message is formed correctly and that all of the required message parts are present in full form. To do this, the filter uses custom policy assertions to make sure that all required message parts are present. The filter also uses the properActionHeader policy assertion to verify that the message contains a WS-Addressing action header. The service can use the properSoapHeader policy assertion to verify that the message contains other SOAP header elements, such as an addressing header and a message ID.

Verifying the XML in the message payload is well-formed and that it conforms to a predefined schema with acceptable data types and ranges of values. The filter uses an XML Schema to validate the contents of the message body. If a specific schema is not required for validation, it uses an XML parser to validate the request body. The filter uses an XML Schema to perform structural validation, data type validation, cardinality of child elements to parent elements, numeric value ranges and regular expression validation for character patterns and ranges. Next step is parsing the request message for malicious content. The service utilizes regular expressions to ensure that the messages contain only valid data. Regular expression validation is implemented in the XSD. The filter uses parameterized SQL queries to access and modify data in databases to mitigate the risk of SQL injection.

Then the service processes the request and responds back to the client. If the request passes all validation checks performed by the message validator, the service processes the message. In this policy, the request size must be checked before any other step. The custom policy assertion must be applied in the pipeline after the message is decrypted but before the request is processed by the service. Regular expression checking in the XML Schema occurs when the request is validated against the message schema in the policy.
assertion. Parameterization of SQL queries occurs when the query is created, prior to execution on the database server.

This implementation pattern uses policy assertions to check for required message parts and to validate the message schema. The following example policy file provides an example of policy assertions for the service.

```xml
<policies xmlns="http://schemas.microsoft.com/wse/2005/06/policy">
  <extensions>
    ...<extension name="bodyValidator" type="MessageValidation.CustomAssertions.BodyValidatorAssertion,MessageValidation.CustomAssertions"/>
  </extensions>
  <policy name="MessageValidationService">
    <bodyValidator xsdPath="Configuration\GetCustomers.xsd"/>
    ...<requireSoapHeader name="MessageID" namespace="http://schemas.xmlsoap.org/ws/2004/08/addressing"/>
    <requireSoapHeader name="To" namespace="http://schemas.xmlsoap.org/ws/2004/08/addressing"/>
    <requireActionHeader/>
  </policy>
  ...</policies>
```

**Figure 5.9 Policy assertions for the service**

In this policy file example, the `<Action>`, `<MessageID>`, and `<To>` elements are required on all incoming request messages. XML document reader method and Exception handler is shown in Figures 5.10 to Figure 5.15.

```csharp
public override void ReadXml(System.Xml.XmlReader reader, IDictionary<string, Type> extensions)
{
    if (reader == null)
        throw new ArgumentNullException("reader");
    if (extensions == null)
        throw new ArgumentNullException("extensions");
    bool isEmpty = reader.IsEmptyElement;
    string xsdPath = reader.GetAttribute("xsdPath");
    if (string.IsNullOrEmpty(xsdPath))
    {
        this.xsdPath = xsdPath;
    }
    else
    {
        throw new ConfigurationErrorsException(Messages.MissingxsdPath);
    }
    reader.ReadStartElement("bodyValidator");
    if (!isEmpty)
        reader.ReadEndElement();
}
```

**Figure 5.10 Custom PolicyAssertion class validator for the received SOAP**
Figure 5.11 Methods to add element and attribute to the XML document

```csharp
public override void WriteXml(System.Xml.XmlWriter writer)
{
    writer.WriteStartElement("bodyValidator");
    writer.WriteAttributeString("xsdPath", this.xsdPath);
    writer.WriteEndElement();
}
```

Figure 5.12 Validator for missing XSD Path

```csharp
public ServiceInputFilter(BodyValidatorAssertion assertion)
{
    string xsdPath = assertion.xsdPath;
    if (!Path.IsPathRooted(xsdPath))
    {
        xsdPath = Path.Combine(AppDomain.CurrentDomain.SetupInformation.ApplicationBase, xsdPath);
    }
    using (StreamReader streamReader = new StreamReader(xsdPath))
    {
        this.schema = XmlSchema.Read(streamReader, ValidationHandler);
        streamReader.Close();
    }
}
```

Figure 5.13 Validator for accounting the exception

```csharp
SoapFilter Methods
public override SoapFilterResult ProcessMessage(SoapEnvelope envelope)
{
    ValidationResults results = new ValidationResults();
    SoapContext.Current.MessageState.Set(results);
    ValidateSchema(envelope.Body.InnerXml);
    if (results.ErrorsCount > 0)
    {
        throw new ApplicationException(string.Format(Messages.ValidationError,results.ErrorMessage));
    }
    return SoapFilterResult.Continue;
}
```

Figure 5.14 Validator of the SOAP body against the specified XML schema (XSD) document

```csharp
/// Performs the validation of the SOAP body against the specified XML Schema (XSD) document.
public void ValidateSchema(string xmlDoc)
{
    try
    {
        XmlReaderSettings settings = new XmlReaderSettings();
        settings.Schemas.Add(this.schema);
        settings.ValidationType = ValidationType.Schema;
        XmlReader reader = XmlReader.Create(new StringReader(xmlDoc),settings);
        // Validate the document.
        while (reader.Read())
        { reader.Close(); }
        catch(Exception ex)
        {
            throw new ApplicationException(string.Format(Messages.SchemaValidationException,ex.Message));
        }
    }
}
```
In the above code, the Messages.MissingXsdPath refers to a resource string that provides a message for the ConfigurationErrorsException that is being thrown. The policy assertion caches the schema in memory that it uses to validate incoming request messages.

### 5.4.2 Coercive Parsing Attack Filter

This filter protects Web Service attacks from intruders by verifying the attack from exception handler and throws exceptions to the client. The exception with corresponding IP address is stored and maintained by the coercive parsing handler. To identify the attacks the following attributes are required to set.

```csharp
if (customHeader.MustUnderstand != true)
    throw new MustUnderstandException("SOAP header entry not understood by processing party ");
if (nameSpace.VersionMismatch != true)
    throw new VersionMismatchException("Invalid namespace defined in SOAP envelope element r");
```

---

**Figure 5.15 Validation counter and validation results method of filter**

In the above code, the Messages.MissingXsdPath refers to a resource string that provides a message for the ConfigurationErrorsException that is being thrown. The policy assertion caches the schema in memory that it uses to validate incoming request messages.
5.4.3 **Oversized Message Attack Filter**

This filter prevents the service from processing request messages that are larger than a specified size. This message validation protects against denial of service attacks, but the message validation must be very efficient when it conducts its validation checks. Otherwise, the message validation logic may be a system bottleneck and may itself become the target of a denial of service attack. Malformed content can include very large messages, in some cases for the purposes of launching a denial of service attack. The filter fixes the maximum message size large enough to allow legitimate messages to be accepted but small enough to prevent attacks. XML message payloads that contain a CDATA field can be used to inject illegal characters that are ignored by the XML parser.

This policy checks any access of local resources of such as CPU. To achieve that it sets the maximum request size in the service’s configuration file to limit the size of messages that the service will process. Then it compares the size of the request to the value established for the maxRequestLength attribute of the `<httpRuntime>` element in the application’s configuration file, which is specified in kilobytes. The maxRequestLength specifies the maximum allowable size for request messages. If the message exceeds this value, the service does not process the message and it returns an error.

To limit the response length of the request, a value for the timeout attribute of the response time element must be set in the service’s Web.config file is shown in Figure 5.16. This value should be set according to the largest response length that it can reasonably expect the service to process. The following XML code set the timeout value for Web Service process.
The service uses a protocol other than HTTP (such as TCP), the filter `<maxMessageLength>` setting is used to limit the size (in kilobytes) of incoming requests is shown in Figure 5.17. The following configuration example shows the `<MaxMessageLength>` set to 1024 KB for a service that uses the SoapClient/SoapService model.

```xml
<configuration>
  ...
  <microsoft.Web.services3>
    ...
    <messaging>
      <maxMessageLength value="1024" />
    </messaging>
    ...
  </microsoft.Web.services3>
  ...
</configuration>
```

5.4.4 Message Replay Attack Filter

The implementation of message replay detection is an identifier is assigned from client side for the message. This identifier provides assurance that the message has not been replayed in transit. Next, the client sends the message to the recipient. The filter verifies the client's identifier and the
message timestamp. The Web Service verifies the message identifier to ensure that the message contents have not been replayed in transit. If the message identifier is valid, then the Web Service compares the message timestamp to its own current clock value. If either the identifier is invalid or the message was received beyond the acceptable time span, the message is rejected. Lastly, the service checks the replay cache for the IdentifierValue field. The Web Service checks the replay cache for the IdentifierValue that is used to uniquely identify the incoming message. If the IdentifierValue is already in the cache, the message is rejected as a duplicate. If the message identifier is not in the cache, the message identifier and timestamp are added to the cache.

Messages cannot be replayed, either accidentally or for malicious intent. Any attempt to replay an intercepted message will result in the message being rejected by the service. Any attempt by the attacker to tamper with the message to spoof the replay mechanism will invalidate the message identifier, which causes the service to reject it.

The Web Service accepts messages sent across a public network from clients that manipulate sensitive data or initiate business processes. The policy has ensured that the Web Service does not process a message that has been intercepted and replayed by an attacker in an attempt to access or manipulate the sensitive data.

This filter prevents the service from allowing and processing messages that have expired, while allowing for clock skew. Also, it prevents the service from accepting and processing messages that intruders have replayed. It supports replay attack detection for Web Services deployed in a Web form through a database-supported replay cache.
The filter uses a custom policy assertion to verify that the service has not previously accepted and processed an incoming message by maintaining a message replay cache. The replay cache is the entity that caches the incoming messages with a unique identifier to detect the replay messages.

The custom policy assertion is implemented by incoming messages are recognized by a message identifier that the policy assertion implements. The message identifier is contained in the <IdentifierValue> element of the message signature. If the message identifier for an incoming message is not in the cache, the service has not processed the message within the lifetime of the cache, and the identifier is added to the cache. If the message identifier is in the cache, the message is rejected as a replayed message.

In detection process, the client assigns an identifier to the message. The client includes a timestamp in the message header and the message using a custom policy assertion to provide data origin authentication. Then the client sends the message to the service. The filter verifies the client’s identifier and the message timestamp.

The filter verifies the freshness of the message by checking the message timestamp. If, after accounting for an acceptable clock skew between the client and service, the message timestamp is older than the server will accept, or the timestamp indicates a future time, the message is rejected. If the message timestamp is valid, the message signature is validated. The service then validates the signature on the message to ensure that it came from an expected client, and its content has not been tampered with while in transit.

The service checks the replay cache for the message identifier; the message identifier is the contents of the <IdentifierValue> element in the message. If the message identifier is already in the cache, the message is rejected as a duplicate. Otherwise, the message identifier is not in the cache,
the message identifier and cache expiration time for the message are added to the cache.

**Service Policy**

The following code example is an example of the configuration for the custom replay detection policy assertion on the service shown in the Figure 5.18.

```xml
<policies xmlns="http://schemas.microsoft.com/wse/2005/06/policy">
  <extensions>
    ...
    <extension name="replayDetection"
    ...
    <policy name="ReplayDetectionService">
      <replayDetection cacheLifeSpanInSeconds="1200" maxMessagePeriodInSeconds="600"/>
    ... </policy>
  </extensions>
</policies>
```

**Figure 5.18 Configuration for the custom replay detection policy assertion on the service**

The replayDetection assertion has two important parameters they are cache life span and maximum message periods configuration parameter.

- cacheLifeSpanInSeconds

  This parameter specifies how long in seconds identifiers will remain in the replay cache. In Figure, this parameter is configured for 1,200 seconds or 20 minutes.

- maxMessagePeriodInSeconds.

  This parameter specifies the maximum message age in seconds that is tolerated by the assertion without accounting for clock skew. In the
preceding example, this parameter is configured for 600 seconds or 10 minutes. The following code configuration snippet provides an example of this setting in the service’s Web.config file. The value is set to 300 seconds is shown in Figure 5.19.

```xml
<microsoft.Web.services3>
    ...
    <security>
        <elapsedTimeInMilliseconds value="300" />  
    </security>
    ...
</microsoft.Web.services3>
```

**Figure 5.19 Code configuration for time tolerance**

A message is accepted or rejected according to policy that takes into consideration the potential time difference between the sender and receiver and an acceptable age for the message to account for longer delays in message transport. The following policy is applied when determining whether to accept an incoming message:

1. The server calculates the message age by subtracting the created value on the message from the current server time. Because of clock skew between the sender and recipient computers, this value can be positive or negative. If the result of this calculation is greater than zero, the message appears to have been created in the past; if the value is less than or equal to zero, it appears to have been created in the future.

2. For a message that appears to have been created in the past or if the server and message creation times are identical, the message will be accepted only when its message age is less than or equal to the values for the `maxMessagePeriodInMilliseconds` parameter plus the `<elapsedTimeInMilliseconds>` setting.
3. For a message that appears to have been created in the future (where the message age is a negative value), the Maximum Message Age setting is not considered, because any delay in message transmission would already have made the message age closer to zero. Instead, the mathematical absolute value of the message age is used. If this value is less than or equal to the Time Tolerance setting, the message is accepted.

Messages are held in the cache for at least as long as the value that is defined in the cacheLifeSpanInSeconds setting. To ensure that the server cannot accept a message after a duplicate message has been removed from the cache, the cacheLifeSpanInSeconds setting must be set to at least the Maximum Message Age + elapsedTime*2.

In the code, the following setting values are configured:

- `<elapsedTimeInSeconds>`. This value is set to the default of 300 seconds or 5 minutes.
- `maxMessagePeriodInSeconds`. This value is set to 600 seconds or 10 minutes.
- `cacheLifeSpanInSeconds`. This value is set to 1,200 seconds or 20 minutes.

These configuration settings are valid because message age plus twice the time tolerance or (600 + (300 x2)) does not exceed the configured cache lifespan of 1,200 seconds.
Replay Cache

The CacheManager class interacts with a replay cache database. Cache expiration is calculated to centralize all policy on the Web Service. The replay cache database table is named ReplayDatabase.

This method provides a solution to prevent the service from processing replayed messages. It does this by rejecting messages that the service has previously received within the valid processing time for them.

Cache Cleaner

The filter must clear the cache at regular intervals to regulate its size. A cache cleanup policy in the filter clears the database cache. The task is scheduled to execute the ClearOldMessages stored procedure at approximately the same interval as the cache life span value configured in the replay detection policy assertion. It executes every 22 minutes to keep the cache reasonably clear.

5.4.5 Semantic URL Attack Filter

This filter guards password resetting while the reset request and response communicated over Internet. Here, the attacker is not appeared, instead the legitimate user tried to steal the password of the other legitimate user in that same Web community. First, the user sends the reset request to the server and then the server sends back the response to the client with corresponding URL with required parameters to reset the password. Now, the legitimate user slightly modifies the parameter to locate the other user’s parameter and receives the password reset access of the other user. In this filter, the mitigation approach is implemented by two significant components of semantic URL filter. Those are random nonce generator, time stamp
calculator. The random nonce generator produces random values and from getting values from identity of task. Afterwards, the nonce value is concatenated with time stamp values to maintain its freshness.

5.5 RESULTS

The injection filter has been configured and embedded in administrator’s login is shown in Figure 5.20. The administrator can enable or disable the filter based on their requirement. This feature will improve the speed of the server.

Figure 5.20 The filter settings in administrators page

5.5.1 Performance Comparison

The proposed system is compared with various existing systems namely input validator, AntiXSS filter, IE explorer, Opera and Firefox. The comparative analysis has been carried out with respect to number of attacks prevented and number of failures as shown in the Figures 5.21 – 5.26.
Figure 5.21 Number of failures for parameter tampering attacks

Figure 5.22 Number of failures for coercive parsing attacks
Figure 5.23 Number of failures for oversized message attacks

<table>
<thead>
<tr>
<th></th>
<th>Proposed system</th>
<th>input validator</th>
<th>Anti XSS filter</th>
<th>IE explorer</th>
<th>Opera</th>
<th>Firefox</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failures</td>
<td>15</td>
<td>200</td>
<td>167</td>
<td>50</td>
<td>102</td>
<td>50</td>
</tr>
<tr>
<td>No. of prevented attacks</td>
<td>985</td>
<td>800</td>
<td>833</td>
<td>950</td>
<td>898</td>
<td>950</td>
</tr>
</tbody>
</table>

Figure 5.24 Number of failures for message replay attacks
Figure 5.25 Number of failures for semantic URL attacks

Figure 5.26 Overall comparative analysis: Number of failures
5.6 SUMMARY

The conventional firewall and filters are lacking to detect and prevent the XML based attacks. Since, these attacks contain huge data and cluttered in nature. The XML based injection filter is proposed to prevent and detect the attacks. All incoming requests are passed through this filter to access the service provider. It is implemented using input validation approach in XML documents. This filter prevents significant classes of attacks, those are parameter tampering filter, coercive parsing filter, oversized message filter, message replay filter and semantic URL filter. This is tested against various XML based attacks to check whether it meets the requirement of the Web Service provider and proved that the proposed approach detects attacks efficiently.