Windows .NET Message Inspector

Overview

This tutorial will show how to protect a simple Windows .NET resource, `GetBalanceService`, using a scope (`accessRestricted`). In the sample we will protect a service which is self-hosted by a console application called DotNetTokenValidator.

First we will define a **Message Inspector** that will help us controlling the incoming request to the `GetBalanceService` resource. Using this Message Inspector we will examine the incoming request and validate that it provides all the necessary headers required by **MobileFirst Authorization Server**.

Prerequesites:

- Make sure to read the Using the MobileFirst Server to authenticate external resources (../) tutorial.
- Understanding of the MobileFirst Platform Foundation security framework (../../).

Jump to:

Create and configure WCF Web HTTP Service

First we will create a **WCF service** and call it `GetBalanceService` which we will protect later by a **message inspector**. In our example we are using a console application as a hosting program for the service.

Here is the code of `getBalance` (the protected resource):

```
public class GetBalanceService : IGetBalanceService {
  public string getBalance()
  {
    Console.WriteLine("getBalance()");
    return "19938.80";
  }
}
```

We should also define a `ServiceContract`:

```
[ServiceContract]
public interface IGetBalanceService
{
  [OperationContract]
  [WebInvoke(Method = "GET",
  BodyStyle = WebMessageBodyStyle.Wrapped,
  ResponseFormat = WebMessageFormat.Json,
  UriTemplate = "getBalance")]
  string getBalance();
}
```

Now that we have our service ready we can configure how it will be used by the host application. This is done in the App.config file as follows:

```
<service behaviorConfiguration="Default" name="DotNetTokenValidator.GetBalanceService">
  <endpoint address="" behaviorConfiguration="webBehavior" binding="webHttpBinding" contract="DotNe
tTokenValidator.IGetBalanceService" />
  <host>
    <baseAddresses>
      <add baseAddress="http://localhost:8732/GetBalanceService" />
    </baseAddresses>
  </host>
</service>
```

Lastly we should run it from the hosting program `Main` method:

```
static void Main(string[] args) {
  // Create the ServiceHost.
  using (ServiceHost host = new ServiceHost(typeof(GetBalanceService)))
  {
    // Enable metadata publishing.
    ServiceMetadataBehavior smb = new ServiceMetadataBehavior();
    smb.HttpGetEnabled = true;

    Console.WriteLine("The service is ready at {0}", host.BaseAddresses[0]);
    host.Open();

    Console.WriteLine("Press <Enter> to stop the service.");
    Console.ReadLine();

    // Close the ServiceHost.
    host.Close();
  }
}
```

> For More information about WCF REST services refer to Create a Basic WCF Web HTTP Service (https://msdn.microsoft.com/en-us/library/bb412178(v=vs.100).aspx)

Define a Message Inspector

Before we dive into the validation process we must create and define a **message inspector** which we will use to protect the resource (the service endpoint). A message inspector is an extensibility object that can be used in the service to inspect and alter messages after they are received or before they are sent. Service message inspectors should implement the `IDispatchMessageInspector` interface:

```
public class MyInspector : IDispatchMessageInspector
```

Any service message inspector must implement the two `IDispatchMessageInspector` methods `AfterReceiveRequest` and `BeforeSendReply`:

```
public class MyInspector : IDispatchMessageInspector {

  public object AfterReceiveRequest(ref Message request, IClientChannel channel, InstanceContext inst
anceContext){
  ...
  }

  public void BeforeSendReply(ref Message reply, object correlationState){
    // In our case there is no need for any code here
  }
}
```

After creating the message inspector it should be defined to protect a certain endpoint. This is done by using behaviors. A **behavior** is a class that changes the behavior of the service model runtime by changing the default configuration or adding extensions (such as message inspectors) to it. This is done using 2 classes: one that configures the message inspector to protect the application endpoint, and the other to return this behavior class instance and type.

```
public class MyCustomBehavior : IEndpointBehavior
{
  ...
  public void ApplyDispatchBehavior(ServiceEndpoint endpoint, EndpointDispatcher endpointDispatcher)
  {
    endpointDispatcher.DispatchRuntime.MessageInspectors.Add(new MyInspector());
  }
  ...
}

public class MyCustomBehaviorExtension : BehaviorExtensionElement
{
  public override Type BehaviorType
  {
    get { return typeof(MyCustomBehavior); }
  }

  protected override object CreateBehavior()
  {
    return new MyCustomBehavior();
  }
}
```

In the `App.config` file we define a `behaviorExtension` and attach it to the behavior class we just created:

```
<extensions>
  <behaviorExtensions>
    <add name="extBehavior" type="DotNetTokenValidator.Inspector.MyCustomBehaviorExtension, DotNe
tTokenValidator"/>
  </behaviorExtensions>
</extensions>
```

Then we add this behaviorExtension to the webBehavior element that is configured in our service as endpoint behavior:

```
<behavior name="webBehavior">
  <webHttp />
  <extBehavior />
</behavior>
```

Message Inspector Implementation

First let's define some constants as class members in our message inspector: MobileFirst server URL, our confidential client credentials and the `scope` that we will use to protect our service with. We can also define a static variable to keep the token received from MobileFirst Authorization server, so it will be available to all users:

```
private const string azServerBaseURL = "http://YOUR-SERVER-URL:9080/mfp/api/az/v1/";
private const string scope = "accessRestricted";
private static string filterIntrospectionToken = null;
private const string filterUserName = "USERNAME"; // Confidential Client Username
private const string filterPassword = "PASSWORD";  // Confidential Client Secret
```

Next we will create our `validateRequest` method which is the starting-point of the validation process that we will implement in our message inspector. Then we will add a call to this method inside the `AfterReceiveRequest` method we mentioned before:

```
public object AfterReceiveRequest(ref Message request, IClientChannel channel, InstanceContext insta
nceContext) {
  validateRequest(request);
  return null;
}
```

Inside `validateRequest` there are 3 main steps that we will implement:

1. **Pre-process validation** - check if the request has an **authorization header**, and if there is - is it starting with the **"Bearer"** prefix.
2. **Get token** from MobileFirst Authorization Server - This token will be used to authenticate the client's token against MobileFirst Authorization Server.
3. **Post-process validation** - check for **conflicts**, validate that the request sent the right **scope**, and check that the request is **active**.

```
private void validateRequest(Message request)
{
  // Pre-process validation: Eextract the clientToken out of the request, check it is not empty and that it start
s with "Bearer"
  string clientToken = getClientTokenFromHeader(request);

  // Get token
  if (filterIntrospectionToken == null)
  {
    filterIntrospectionToken = getIntrospectionToken();
  }

  // Check client auth header against mfp authrorization server using the token I received in previous step
  HttpWebResponse introspectionResponse = introspectClientRequest(clientToken);

  // Check if introspectionToken has expired (401)
  // - if so we should obtain a new token and resend the client request
  if (introspectionResponse.StatusCode == HttpStatusCode.Unauthorized)
  {
    filterIntrospectionToken = getIntrospectionToken();
    introspectionResponse = introspectClientRequest(clientToken);
  }

  // Post-process validation: check that the MFP authrorization server response is valid and includes the req
uested scope
  postProcess(introspectionResponse);
}
```

Pre-process Validation

The pre-process validation is done as part of the getClientTokenFromHeader() method. This process is based upon 2 checks:

1. Check that the authorization header of the request is not empty.
2. If it is not empty - check that the authorization header starts with the "Bearer " prefix.

In both cases we should respond with an **Unauthorized response status** (401) and add the **WWW-Authenticate:Bearer** header.
After validating the authorization header this method returns the token received from the client application.

```
private string getClientTokenFromHeader(Message request)
{
  string token = null;
  string authHeader = null;

  // Extract the authorization header from the request
  var httpRequest = (HttpRequestMessageProperty)request.Properties[HttpRequestMessageProperty.Nam
e];
  authHeader = httpRequest.Headers[HttpRequestHeader.Authorization];

  // Pre-process validation
  if ((string.IsNullOrEmpty(authHeader) || !authHeader.StartsWith("Bearer", StringComparison.CurrentCult
ure)))
  {
    WebHeaderCollection webHeaderCollection = new WebHeaderCollection();
    webHeaderCollection.Add(HttpResponseHeader.WwwAuthenticate, "Bearer");
    returnErrorResponse(HttpStatusCode.Unauthorized, webHeaderCollection);
  }

  // extract the token without the "Bearer " prefix
  try {
    token = authHeader.Substring("Bearer ".Length);
  }
  catch (Exception ex) {
    Console.WriteLine(ex);
  }

  return token;
}
```

`returnErrorResponse` is a helper method that receives an httpStatusCode and a WebHeaderCollection, prepares the response and sends it back to the client application. After sending the response to the client application it completes the request.

```
private void returnErrorResponse(HttpStatusCode httpStatusCode, WebHeaderCollection headers)
{
  OutgoingWebResponseContext outgoingResponse = WebOperationContext.Current.OutgoingResponse;
  outgoingResponse.StatusCode = httpStatusCode;
  outgoingResponse.Headers.Add(headers);
  HttpContext.Current.Response.Flush();
  HttpContext.Current.Response.SuppressContent = true; //Prevent sending content - only headers will be
sent
  HttpContext.Current.ApplicationInstance.CompleteRequest();
}
```

Obtain Access Token from MobileFirst Authorization Server

In order to authenticate the client token we should **obtain an access token as the message inspector** by making a request to the **token endpoint**. Later we will use this received token to pass the client token for introspection.

```
private string getIntrospectionToken()
{
  string returnVal = null;
  string strResponse = null;

  string Base64Credentials = Convert.ToBase64String(
    System.Text.ASCIIEncoding.ASCII.GetBytes(
      string.Format("{0}:{1}", filterUserName, filterPassword)
    )
  );

  // Prepare Post Data
  Dictionary<string, string> postParameters = new Dictionary<string, string> { };
  postParameters.Add("grant_type", "client_credentials");
  postParameters.Add("scope", "authorization.introspect");

  try {
    HttpWebResponse resp = sendRequest(postParameters, "token", "Basic " + Base64Credentials);
    Stream dataStream = resp.GetResponseStream();
    StreamReader reader = new StreamReader(dataStream);
    strResponse = reader.ReadToEnd();

    JToken token = JObject.Parse(strResponse);
    returnVal = (string)token.SelectToken("access_token");
  }
  catch (Exception ex) {
    Debug.WriteLine(ex);
  }

  return returnVal;
}
```

The `sendRequest` method is a helper method that is responsible for sending requests to MobileFirst Authorization server.
It is being used by `getIntrospectionToken` to send a request to the token endpoint, and by `introspectClientRequest` method to send a request to the introspection endpoint. This method returns an `HttpWebResponse` which we use in `getIntrospectionToken` method to extract the access_token from and store it as the message inspector token. In `introspectClientRequest` method it is used just to return the MFP authorization server response.

```
private HttpWebResponse sendRequest(Dictionary<string, string> postParameters, string endPoint,
string authHeader) {
  string postData = "";
  foreach (string key in postParameters.Keys)
  {
    postData += HttpUtility.UrlEncode(key) + "=" + HttpUtility.UrlEncode(postParameters[key]) + "&";
  }

  HttpWebRequest request = (HttpWebRequest)WebRequest.Create(new System.Uri(azServerBaseURL
+ endPoint));
  request.Method = "POST";
  request.ContentType = "application/x-www-form-urlencoded";
  request.Headers.Add(HttpRequestHeader.Authorization, authHeader);

  // Attach Post Data
  byte[] data = Encoding.ASCII.GetBytes(postData);
  request.ContentLength = data.Length;
  Stream dataStream = request.GetRequestStream();
  dataStream.Write(data, 0, data.Length);
  dataStream.Close();

  return (HttpWebResponse)request.GetResponse();
}
```

Send request to Introspection Endpoint with client token

Now that we are authorized by MobileFirst Authorization Server we can **validate the client token** content. We send a request to the **Introspection endpoint**, adding the token we received in the previous step (`filterIntrospectionToken`) to the request header and the client token in the post data of the request. Next we will examine the response from MobileFirst Authorization Server in `postProcess` method.

```
private HttpWebResponse introspectClientRequest(string clientToken) {
  // Prepare the Post Data - add the client token to the postParameters dictionary with the key "token"
  Dictionary<string, string> postParameters = new Dictionary<string, string> { };
  postParameters.Add("token", clientToken);

  // send the request using the sendRequest() method and return an HttpWebResponse
  return sendRequest(postParameters, "introspection", "Bearer " + filterIntrospectionToken);
}
```

Post-process Validation

Before proceeding to the `postProcess` method we want to make sure that the response status is not **401 (Unauthorized)**.
401 (Unauthorized) response status at this point indicates that the message inspector token (`filterIntrospectionToken`) has expired. If the response status is 401 (Unauthorized) then we call `getIntrospectionToken` to get a new token for the message inspector and call `introspectClientRequest` again with the new token.

```
if (introspectionResponse.StatusCode == HttpStatusCode.Unauthorized)
{
  filterIntrospectionToken = getIntrospectionToken();
  introspectionResponse = introspectClientRequest(clientToken);
}
```

The main purpose of the postProcess method is to examine the response we received from MobileFirst Authorization Server, but before extracting and checking the response, we must **make sure that the response status is 200 (OK)**. If the response status is **409 (Conflict)** we should forward this response to the client application, otherwise we should throw an exception.
If the response status is 200 (OK) then we initialize the `AzResponse` class, which is a class defined to reprisent the MobileFirst Authorization Server response, with the current response. Then we check that the **response is active** and that it includes the right **scope**:

```
private void postProcess(HttpWebResponse introspectionResponse)
{
  if (introspectionResponse.StatusCode != HttpStatusCode.OK) // Make sure that HttpStatusCode = 200 ok
(before checking active==true & scope)
  {
    if (introspectionResponse.StatusCode == HttpStatusCode.Unauthorized) // We have a real problem sin
ce we already obtained a new token
    {
      throw new WebFaultException<string>("Authentication did not succeed, Please try again...", HttpStat
usCode.BadRequest);
    }
    else if (introspectionResponse.StatusCode == HttpStatusCode.Conflict) // Check Conflict response (40
9)
    {
      returnErrorResponse(HttpStatusCode.Conflict, introspectionResponse.Headers);
    }
    else
    {
      throw new WebFaultException<string>("Authentication did not succeed, Please try again...", HttpStat
usCode.BadRequest);
    }
  }
  else
  {
    AzResponse azResp = new AzResponse(introspectionResponse); // Casting the response to an object
    WebHeaderCollection webHeaderCollection = new WebHeaderCollection();

    if (!azResp.isActive)
    {
      webHeaderCollection.Add(HttpResponseHeader.WwwAuthenticate, "Bearer error=\"invalid_token\"");
      returnErrorResponse(HttpStatusCode.Unauthorized, webHeaderCollection);
    }
    else if (!azResp.scope.Contains(scope))
    {
      webHeaderCollection.Add(HttpResponseHeader.WwwAuthenticate, "Bearer error=\"insufficient_scope
\", scope=\"" + scope + "\"");
      returnErrorResponse(HttpStatusCode.Forbidden, webHeaderCollection);
    }
  }
}
```

Sample

Download the .NET message inspector sample (https://github.com/MobileFirst-Platform-Developer-Center/DotNetTokenValidator/tree/release80).

Sample usage

1. Use Visual Studio to open, build and run the sample as a service (run Visual Studio as an administrator).
2. Make sure to update the confidential client (../#confidential-client) and secret values in the MobileFirst Operations Console.
3. Deploy either of the security checks: **UserLogin (../../user-authentication/security-check/)** or **PinCodeAttempts (../../credentials-validation/security-check/)**.
4. Register the matching application.
5. Map the `accessRestricted` scope to the security check.
6. Update the client application to make the `WLResourceRequest` to your servlet URL.