Web application defense at the gates – Leveraging IHttpModule

Abstract

Web applications are vulnerable to many attacks, mainly due to poor input validation at the source code level. Firewalls can block access to ports but once a web application goes live and TCP ports 80 and 443 are accessible, the web application can be an easy prey for attackers. *HTTP traffic is legitimate traffic* for web applications; all the more reason to include application-level content-filtering over unencrypted and encrypted communication channels. Application-level content filtering is possible to some extent but may not work over *HTTPS* (port 443). The only way to provide a strong defense is by applying powerful content-filtering at the application-level for both TCP port 80 and TCP port 443.

The .Net framework with ASP.NET provides the *IHttpModule* interface access to HTTP pipes – the lowest of programming layers – before an incoming HTTP request hits the web application. This can provide defense at the gates. In this paper, we look at how one can build this sort of defense in all three aspects – coding, deployment and configuration.

Shreeraj Shah

Co-Author: "Web Hacking: Attacks and Defense" (Addison Wesley, 2002) and published several advisories on security flaws.

n e t - s q u a r e
http://www.net-square.com
shreeraj@net-square.com

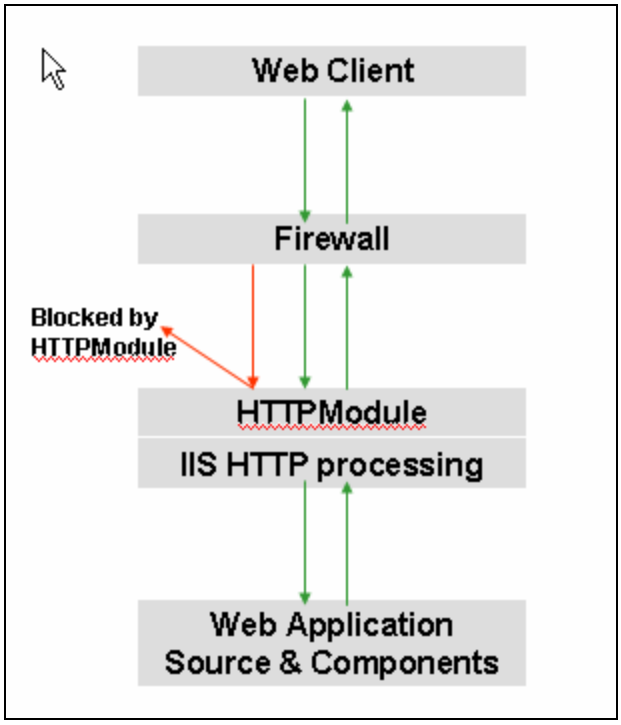
Table of Contents

Acknowledgement

Lyra Fernandes for her help on documentation.

Introduction

Traditional firewalls block traffic at defined ports but are limited in their ability to provide content filtering capabilities. Web applications usually run on TCP ports 80 or 443. Access to these ports is open at firewalls. Without security controls for content filtering and input sanitization in place, distinguishing between legitimate HTTP traffic and potentially malicious requests camouflaged as legitimate HTTP traffic on both encrypted and unencrypted communication channels, is extremely difficult. Poor input validation has been the root cause of security breaches on many web applications running on the Internet in today's world.



This problem was identified and primitive content filtering was provided at the firewall level but it was unable to block encrypted traffic going back and forth over 443. All critical applications such as banking, financial or payment gateways run on port 443. The scope of the problem is reduced but not eliminated entirely. IIS web server provides ISAPI extensions to handle incoming HTTP requests. A similar feature is available with Apache as well. Microsoft released a tool URLScan which provides services-level content filtering but it is not powerful enough to fine tune defense at the application-level.

Microsoft's .Net framework includes two interfaces – *IHTTPModule* and *IHTTPHandler*. These two interfaces can be leveraged to provide application-level defense customized to application-level, folder-level or variable-level. This can act as the first line of defense, before any incoming request touches the web application source code level. This is web application defense at the gates, for the .Net framework on IIS. This paper discusses how to provide using HTTPModule. A similar concept can be extended to *HTTPHandler.*

NOTE: The sample code shown here is written in C#. You must create a project as "Class Library" since you will be creating a .dll file that fits into the IIS HTTP processing chain or pipe. "System.Web" must be included as reference assembly to the project. The IHTTPModule interface resides in "System.Web".

Building up WebAppMod using IHttpModule Interface

Definition and code Initialization

The WebAppMod namespace is created which in turn, hosts the WebAppWall class by extending the IHttpModule interface. With this we can access events of HTTP pipe from the top since the IHttpModule interface is higher in the pipe than any other handler accessing incoming HTTP requests.

[Code Snippet]

```
using System;
using System.Web;
using System.Text.RegularExpressions;
namespace WebAppMod
{
      public class WebAppWall : IHttpModule
      {
```

Regex utility function for support

Regular expressions ("regex") are sets of symbols and syntactic elements used to match patterns of text. They allow more complex search and replace functions to be performed in a single operation.

In our example, we need to filter HTTP input requests that contain metacharacters that could break a web application, disclosing enough useful information to an attacker. We do this by using a supporting Regex function to process regular expressions. Take a look at the code snippet below:

[Code Snippet]

```
public string[] setPattern(string doc,string pat,int num)
{
      Regex exp = new Regex(@pat,RegexOptions.IgnoreCase);
      MatchCollection mc = exp.Matches(doc);
      string[] results = new string[mc.Count];
      for (int i=0;i<mc.Count;i++)
      {
            Match FirstMatch = mc[i];
            results[i] = FirstMatch.Groups[num].ToString();
      }
      return results;
}
```

The function *Regex* takes three parameters as input –

- ✍ doc, which is the target string to search for a pattern,
- ✍ pat, which is the set of characters to be matched in the target string and
- ✍ num, which is the match number

This function will return an entire array of strings – *results* – with all instances of matched patterns.

Accessing Init of IHttpModule

[Code Snippet]

```
public void Init(HttpApplication httpApp)
{
      httpApp.BeginRequest += new EventHandler(this.OnBeginRequest);
```

The above lines of code will get called as part of the Init (IHttpModule) interface. An HttpApplication handler object is provided for processing. This object has an event called "BeginRequest" which will be invoked before an HTTP request is trapped by your web application, triggering an event where a "`ProcessRequest`" function gets invoked.

[Code Snippet]

```
string[] query,post;
public void Init(HttpApplication App)
{
      App.BeginRequest += new EventHandler(this. ProcessRequest);
      string inifile = Environment.CurrentDirectory +
                                    "\\webappmod\\webappmod.ini";
      System.IO.StreamReader reader = new
                                    System.IO.StreamReader(inifile);
      string data = reader.ReadToEnd ();
      reader.Close();

      string[] qres = setPattern(data,"<QUERY>(.*?)</QUERY>",1);
      query = new string[qres.Length];
      query = qres;

      string[] pres = setPattern(data,"<POST>(.*?)</POST>",1);
      post = new string[pres.Length];
      post = pres;
}
```

Major input points for web applications are:

1. Querystring
2. POST buffer

Information sent from a form with the GET method, also called *querystring*, is appended to the end of a resource by a question mark (?) and displayed in the browser's address bar. A POST buffer, on the other hand, uses the POST method for form submissions. We need to provide critical input validation at these two places.

> ✍ If the method is "GET", the user agent takes the value of action, appends a **?** to it, then appends the form data set. The user agent then traverses the link to this URI.
>
> ✍ If the method is "POST", the user agent conducts an HTTP post transaction using the value of the action attribute and a message created according to the content type specified by the enctype attribute.

Example of a *Querystring*:

```
http://192.168.131.3/dvds4less/details.aspx?id=1
```

Example of a *POST buffer*:
HTTP request would look like this once form submission is completed.

```
POST /dvds4less/checkout_form.aspx HTTP/1.1
Host: 192.168.131.3
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.0; en-US;
rv:1.7.3) Gecko/20040910
Accept:
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/
plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://192.168.131.3/dvds4less/cart.aspx?id=1&quantity=1
Cookie: ASP.NET_SessionId=0zrvzp45nzb1sj45piri0f55
Content-Type: application/x-www-form-urlencoded
Content-Length: 60

product_id_0=1&quantity_0=1&order_num=513745&submit=Checkout
```

To achieve the above objective we have created two string arrays – one for a *querystring* (query) and the other for a *post buffer* (post). These two string arrays are going to be filled up from webappmod.ini file; a part of the system directory on Windows.

For example, if *C:\Winnt\System32* is a system directory on your system, it must have a folder called webappmod and a file called webappmod.ini. We grab the value of the environment variable "`Environment.CurrentDirectory`" and append it to our folder.

```
string[] res = setPattern(data,"<QUERY>(.*?)</QUERY>",1);
res = setPattern(data,"<POST>(.*?)</POST>",1);
```

The above two lines are responsible for fetching XML tags defined for Query and POST. These multiple tags are processed and the resulting pattern consolidated in respective arrays. These patterns are the *rule sets* for input sanitization. In other words, rejection of input is done on the basis of these criteria.

You may have noticed that these arrays are global in scope, making them accessible to other functions as well.

Processing incoming request using ProcessRequest function

An instance of the HttpApplication object is created and passed from "BeginRequest" event in the chain.

[Code Snippet]

```
public void ProcessRequest (object o, EventArgs ea)
      {
            HttpApplication app = (HttpApplication) o;
```

The above line of code is invoked every time an HTTP request is directed at your application.

[Code Snippet]

```
string querystring = app.Request.ServerVariables["QUERY_STRING"];

if(query.Length > 0)
{
      for(int j=0;j<query.Length;j++)
      {
              string[] q = setPattern(querystring,query[j],0);
              if(q.Length>0)
              {
                  app.Response.Write("Security Error");
                  app.Response.End();
              }
      }
}
```

In the above code we grab the querystring from the server variables of the request object and process them using our list of objectionable patterns defined in our query variable. If a pattern match is found we throw a security error and terminate the response, otherwise we allow the request to go through.

Similar action is carried out for the POST buffer below:

[Code Snippet]

```
string postreq = "";
if(app.Request.ServerVariables["REQUEST_METHOD"] == "POST")
{
      long streamLength = app.Request.InputStream.Length;
      byte[] contentBytes = new byte[streamLength];
      app.Request.InputStream.Read(contentBytes, 0, (int)streamLength);
      postreq = System.Text.Encoding.UTF8.GetString(contentBytes);
      app.Request.InputStream.Position = 0;
```

In above code snippet we access Inputstream of the request object and fetch the POST buffer of the incoming request.

Next, we compare all patterns of the POST string array with the HTTP POST buffer received. Any objectionable pattern found, results in termination of the response after a security error message is displayed. A legitimate request will go through to the web application.

[Code Snippet]

```
if(post.Length > 0)
{
      for(int k=0;k<post.Length;k++)
      {
            string[] p = setPattern(postreq,post[k],0);
            if(p.Length>0)
            {
                  app.Response.Write("Security Error");
                  app.Response.End();
            }
      }
}
```

Compile the above code to get an ***assembly*** called ***WebAppMod.dll***. Deploy this assembly for web application defense at the gates.

Deploying WebAppMod.dll in the environment

Step 1: Create webappmod.ini

First, create a *webappmod.ini* file in format shown below. Place this file in the webappmod folder residing in the machine's system directory.

[webappmod.ini]

```
<QUERY>
   id=(.*?['\"%*$#@]|.*?(select|exec|update))[^&]*([&]|$)
</QUERY>

<QUERY>
   quantity=(.*?['\"%*$#@]|.*?(select|exec|update))[^&]*([&]|$)
</QUERY>

<POST>id=(.*?['\"%*$#@]|.*?(select|exec|update))[^&]*([&]|$)</POST>
<POST>quantity=(.*?['\"%*$#@]|.*?(select|exec|update))[^&]*([&]|$)</POST>
```

We will see the purpose of these patterns in detail in subsequent sections.

Step 2: Create a "bin" folder

Create a bin folder in your application directory (virtual root or virtual site) on the IIS web server and place the WebAppMod.dll file in this folder.

Step 3: Modify web.config

Next, add the following lines to your *web.config* file

```
<httpModules>
    <add type="WebAppMod.WebAppWall, WebAppMod" name="WebAppWall" />
</httpModules>
```

This will load webappmod at server startup or when the assembly file WebAppMod.dll is changed.

Defending web application at the gates by configuration

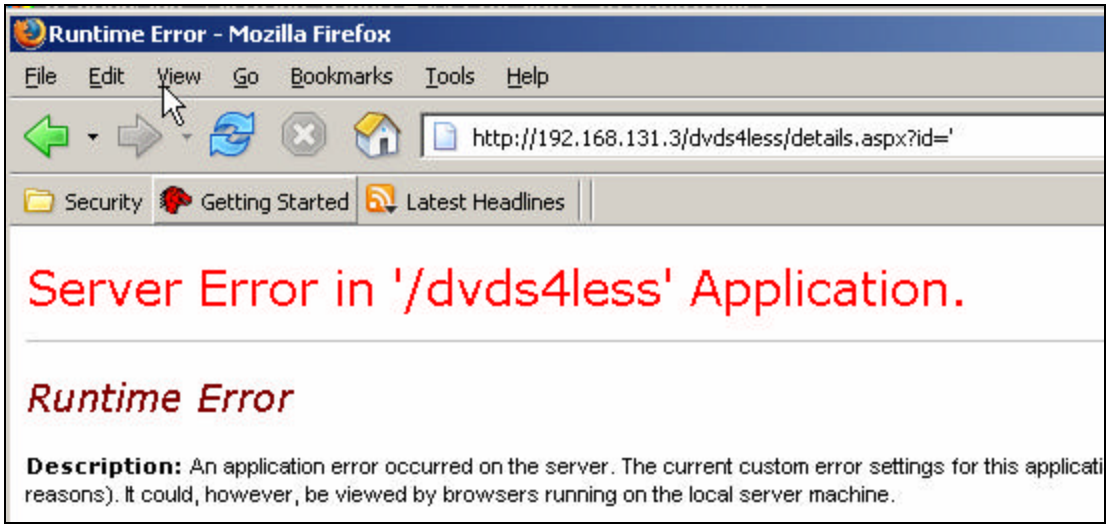
Let us look at a web application that takes two parameters as input – id and quantity. We have already created a webappmod.ini file in the previous section. Now let's see how we can defend this variable called "id".

For example, in the URL,
http://192.168.131.3/dvds4less/details.aspx?id=1

id=1 is the querystring that is passed to the web application code. The web application code reads in this value and displays the details accordingly. So far so good.
Now, what if some one passes malicious content such as this:

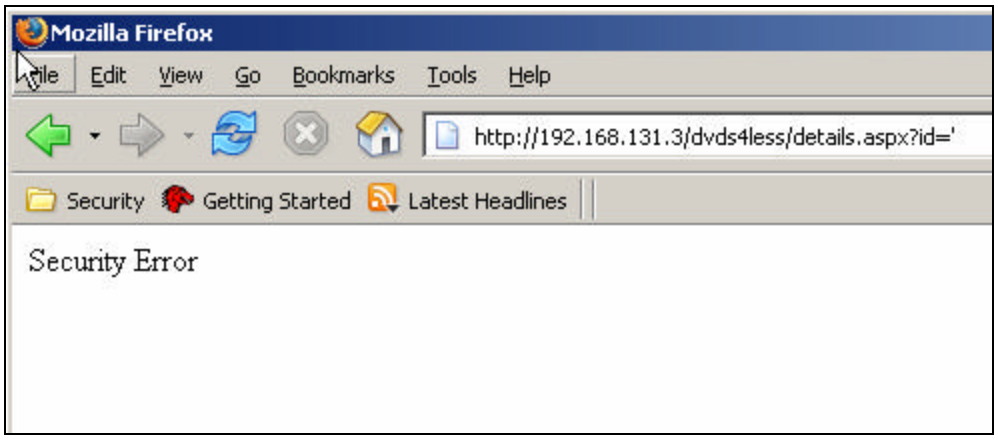
http://192.168.131.3/dvds4less/details.aspx?id='

id=' (Single quote) breaks the application and we get an error page like the one shown below, before loading webappmod.



This error is generated from within the web application; the attack touches the application code.

Let us load *WebAppMod.dll*, pass the same HTTP request and view the results. We still get an error, but there's a difference.



A Security Error is generated; only this time, it is generated from WebAppMod.dll and not from within the Web Application. The web application code is completely isolated and neither the code nor the components are directly affected by the above request. True defense at the gates of web applications.

Regex Magic

Let's see how regex magic is performed from a *.ini* file.

[Sample regex for variable id]

```
<QUERY>id=(.*?['\"%*$#@]|.*?(select|exec|update))[^&]*([&]|$)</QUERY>
```

Let's slice up the above pattern and see how it works:

1. We look for "**`id=`**" which specifies the variable linked to the application layer.
2. **`.*?['\"%*$#@]`**
   this pattern will grab metacharacters that are likely to break the application. *Single quote, %, $,* etc.
3. **`.*?(select|exec|update)`**
   this pattern looks for SQL special words which are responsible for a *sql injection* attack.
4. **`|`** signifies an **`(or)`**. The pattern will perform an OR operation on either of clauses 2 or 3 mentioned above.
5. **`[^&]*([&]|$)`**
   this pattern will make sure if the *id* variable is at the start of the line, at the end of line or in between, it is segregated by the **&** character. This character is a delimiter, responsible for variable bifurcation in the Querystring or POST buffer of the HTTP protocol.

The same pattern will work either in a querystring or in the POST buffer. In this manner, one can craft required regex patterns and formulate an INI file. This allows an application defense to be fine-tuned at the variable level.

Conclusion

ASP.NET offers an extensible framework for server-side HTTP programming at a lower-level infrastructure.

This paper introduces the *IHttpModule* interface – the lowest of programming layers - provided with ASP.Net and shows how this interface can be used to securely and efficiently implement HTTP request processing in an ASP.NET-based web application.

This paper will enhance your vision on the usage of the IHttpModule interface in web applications developed in ASP.NET and is meant to serve as a stepping-stone to add sophisticated functionality such as application-level content filtering, to defend web applications at the gates.