

Forum:[Hack Forums](#)

Thread: [\[TUTORIAL\] Non-conventional WAF/IDS evasion techniques \[PART 2\]](#)

M L T - December 10, 2018, 6:25 am

Non-Conventional WAF/IDS Evasaion techniques - Part 2:

Introduction: This is part 2 of my WAF/IDS evasion tutorial. This tutorial series covers non-conventional methods of WAF evasion as opposed to traditional signature evasion techniques. If you haven't read part 1 already, then [read it now](#) before coming back here.

Manipulation of parameter names: Parameter names can be manipulated in various ways, depending upon the server-sided languages running on the vulnerable server. In this post I will be giving examples of both PHP and ASP, followed by an explanation as to why manipulation of parameter names within a URL can result in a valid bypass.

This is especially useful for bypassing virtual patching, for example similar to that which can be often applied to mod_security with a hardcoded path as part of the 'patch' — ability to manipulate this hardcoded value can result in a bypass. The + symbol in PHP can be used to achieve this, whereas in ASP the % symbol will achieve a similar result.

Within ASP, invalid URL encoding can be added to parameter names and it will “clean up” for you and remove any invalid encoding. Note that the encoding does have to be invalid for this to work:

```
/path/vuln.asp?%value=payload  
/path/vuln.aspx?%}9value=payload
```

In PHP, the plus sign can be used to achieve the same effect: /path/vuln.php?+value=payload

Here, PHP will clean up in a similar sense to that of ASP. The purpose here is to create what is still a valid path and parameter name, while not matching the path hardcoded within the virtual patch. These techniques can be combined with the path manipulation section listed further on in this tutorial.

CRLF and Control Characters: Certain control characters can cause WAF's to act in unexpected ways, in some cases a mere nullbyte (%00) is enough to stop them from working entirely — although it should be noted that any WAF worth its salt will not only remain unaffected by an injected nullbyte or null terminator, but will also have signatures in place to cause the input of such characters to result in a WAF response. That being said, some control characters can often yield fruitful results.

To see how the server handles nullbytes, try adding one before the extension to a file that exists on the server, for example, change:

Quote:<http://example.com/file.txt>

to:

Quote:<http://example.com/file%00.txt>

In addition to nullbytes, the percent-encoded equivalents for CR/LF and other similar control characters can trigger unexpected results regarding some WAF software. Just append them to the end of your payload, I've seen it working with the following characters:%0d (CR)

%0a (LF)

%0d%0a (CRLF)

%09 (TAB)

%0B

%0

Although this next part is slipping into the realm of signature evasion (which I said I would avoid in this guide), I figured it's still worth mentioning anyway. These characters can also be used for splitting your payload, for example, take the following payload (assume that 'javascript' is blacklisted):

Quote:javascript:alert(1);

CR/LF characters could be used like so:

Quote:j%0aa%0av%0aa%0as%0ac%0ar%0ai%0ap%0at:alert(1);

HTTP Parameter Pollution: I'm assuming those reading this already have an understanding of HTTP parameter pollution, but for those who aren't, I will offer a quick explanation.

HTTP parameter pollution is the name used to describe the unexpected behaviour that can result from the same HTTP parameter being supplied multiple times within an application. This can refer to a multitude of security vulnerabilities, ranging from the ability to make modifications to internal variable values, to full-blown authentication bypass. The severity of HTTP parameter pollution is entirely dependant on the unique nature of the application and the actions that are generated as a result of the pollution of HTTP parameters.

There are a few primary ways in which HTTP parameter pollution can be used to circumvent a WAF. The first way, is just simply including the payload twice (once in each parameter), like so:

Quote:<http://example.com/vuln.php?page=cat> /etc/passswd&page=cat /etc/passwd

A variation of this would be to include the parameter twice, but only contain the payload within one of the parameters while having no value assigned to the other:

Quote:<http://example.com/vuln.php?page=cat> /etc/passswd&page

<http://example.com/vuln.php?page=cat> /etc/passswd&page=

Or assign an arbitrary value:

Quote:<http://example.com/vuln.php?page=cat> /etc/passswd&page=blahblahasdfgh

The reasoning here is that it attempts to validate the empty 'page' parameter while allowing the other identically named parameter to slip through without detection.

One effective bypass technique involving HTTP parameter pollution is to split your payload into

multiple parts (if permitted), here is an example:

Quote:<http://example.com/vuln.php?page=cat> &page=/etc/passwd
<http://example.com/vuln.php?page=cat> &page=/etc&page=/passwd

HTTP Verb Tampering: Remember that your request does not necessarily need to be sent with a valid HTTP verb. Malformed verbs can work fine, for example, the request could be sent via 'GETS' or 'LOLOL' rather than 'GET' and would still run as expected in many cases. People expect using malformed HTTP verbs to result in an error, but this is not what happens in many cases.

Using uncommon HTTP verbs is also a possibility, especially in cases where the WAF or IDS is only monitoring POST or GET requests. For example, if your malicious GET request is triggering a WAF response, then try sending that same request via HEAD or PUT wherever possible. For example, if you are trying to initiate a GET request by loading the vulnerable site in your address bar with your payload stored in URL-based GET parameters, then try something like the following instead:

Quote:PUT /path/vuln.php?value=PAYLOAD HTTP/1.1
host: www.example.com

If the WAF is being applied to your GET request, try sending it via POST or vice-versa. For example, if you had a vulnerable URL parameter (GET) that was triggering the WAF, try sending a POST request too (alongside the URL to the vulnerable script, but containing no GET parameters in the URL)

Within PHP specifically, depending on configuration, cookie values can be treated as parameters. So for example, if you had the following vulnerability:

Quote:<http://example.com/vuln/vuln.php?cmd1=;cat> /etc/passwd

Assume the WAF was blacklisting your GET requests that you were supplying as part of the URL. You could call the parameter through a cookie value, by setting a cookie like so:

Quote:GET /vuln/vuln.php
Cookie: cmd1=;cat /etc/passwd

HTTP Request Smuggling:

HTTP request smuggling, as the aptly-named terminology would suggest, is a method of smuggling HTTP requests which can result in detection being avoided.

This attack vector is achieved by an attacker modifying a HTTP request as it is on its route to its destination. Such a vulnerability exists as a direct result of improper handling of deformed inbound HTTP requests. I will not be going into detail here (although you will find a detailed explanation on our upcoming wiki) — this would require a blog post of its own to explain it in detail along with the possible outcomes (web cache poisoning, XSS, session fixation, and more). The reason it's being mentioned here is because it allows an attacker to 'smuggle' a request to one device without the other device being aware of it. Meaning if properly crafted, an attacker can 'smuggle' a request to the target server while slipping through the WAF or IDS undetected. I would definitely suggest researching this technique and its applications for WAF/IDS evasion. Despite me not going into detail (due to this needing a blog post of its own), a simple Google search alongside a primitive understanding of how HTTP works as a protocol should explain to you why this is so useful for bypassing WAF or IDS software. For the readers of this guide, your homework is to read up on HRS (HTTP Request Smuggling) and attempt to pull off a real-world attack that allows you to bypass a

firewall (legally, of course!)