Bugtraq: Whitepaper by Amit Klein: "HTTP Response Smuggling"

Bugtraq: Whitepaper by Amit Klein: "HTTP Response Smuggling" - Amit Klein, seclists.org.

- Published: date not stated
- Original: <https://seclists.org/bugtraq/2006/Feb/364>
- Preserved from: https://seclists.org/bugtraq/2006/Feb/364 (live) on 2026-09-10
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

[[image: bugtraq logo]](https://seclists.org/bugtraq/)

Bugtraq mailing list archives

Whitepaper by Amit Klein: "HTTP Response Smuggling"

From: "Amit Klein (AKsecurity)" <aksecurity () hotpop com> *Date*: Mon, 20 Feb 2006 21:25:14 +0200

HTTP Response Smuggling

Or "HTTP Response Splitting is [still] Mostly Harmful" ;-)

Amit Klein, February 2006

Introduction

============

Recently, several anti- HTTP Response Splitting strategies has been suggested and/or put to use by various individuals and vendors. Apparently, those individuals and vendors did not subscribe to the somewhat strict approach recommended in [1], which is, to simply disallow CR and LF in data embedded in HTTP response headers. Rather, the recent anti-HTTP Response Splitting suggestions attempt to take a more granular approach. However, it seems that unfortunately, this approach is basically flawed, because it does not take into account variations and tolerance in the parsing of HTTP responses among proxy servers and clients. This paper presents HTTP Response Smuggling - a way to evade those anti- HTTP response splitting strategies. HTTP Response Smuggling makes use of HTTP Request Smuggling -like techniques ([2]) to exploit the discrepancies between what an anti- HTTP Response Splitting mechanism would consider to be the HTTP response stream, and the response stream as parsed by a proxy server (or a browser).

Technique #1 - Who needs a CRLF anyway?

=======================================

In [3] and [4], it seems that the major defense line against HTTP Response Splitting is disallowing the CRLF sequence ([4] recommends also disallowing the string "HTTP/1.", as well as other strings - this will be covered below). Apart from the serious false positive problem this inflicts (forms with TEXTAREA fields expect multi-line submission, which has CRLF in it), it is also quite ineffective against HTTP Response Splitting.

Many proxy servers (e.g. Apache 2.0.55 mod_proxy and Sun Java System Web Proxy Server 4.0, DeleGate 8.11.5) simply allow LF where CRLF is expected. This is also true for Microsoft IE 6.0 SP2 and Mozilla Firefox 1.5. As such, an HTTP Response Splitting attack can be devised containing LFs only (and was indeed demonstrated on Apache 2.0.55 mod_cache+mod_proxy). Note that treating LF as an end of line marker is in violation of the "strict" RFC 2616 [5] section 2.2, which defines the CRLF sequence as the end of line marker, yet at the same time, the RFC (in section 19.3) recommends parsing LF as CRLF.

Poisoning the cache of Apache 2.0.55 and Sun Java System Web Proxy Server 4.0 (see appendix) succeeded when only LFs were used.

Technique #2 - The oldest trick in the Smuggling book
===

In [6], the author suggest anti- HTTP Response Splitting technique based on the server marking where it considers the start of headers and end of headers are (using a marker such as a random string which is unknown to the attacker at the injection time). The HTTP client (proxy or browser) then has to verify that the start of headers and end of headers markers match. Putting aside usability issues such as header reordering (note that the RFC [5] section 4.2 states that "The order in which header fields with differing field names are received is not significant.", meaning that RFC compliant implementations are not required to maintain order among different headers, and indeed some are known to reorder headers), the fact of the matter is that still, some HTTP Response Splitting attacks are possible. In this case, the double Content-Length technique (a classic smuggling trick) comes in handy. Let us assume that the injection point occurs before the original Content-Length in the headers section. In such case, the attacker injects a Content-Length header of his/her own. As it happens, Microsoft IE 6.0 SP2 and Apache 2.0.55 mod_proxy will use the first Content-Length header, and ignore any additional Content-Length headers (while Mozilla Firefox 1.5, Sun Java System Web Proxy Server 4.0 and Delegate 8.11.5 will use the last Content-Length header, and ignore any preceding headers - so if the injection point occurs after the original Content-Length header, they can be exploited).
The injected Content-Length header terminates the first request at a location of the attacker's choice. The attacker needs to carefully choose this location to point at another injection point (this time in the response body) in which he/she can embed a complete HTTP response, including a spoofed start of headers marker and end of headers marker. This second injection is an additional requirement, and as such, arguably limits the attack, however - there are cases wherein a second injection is native to

the situation (see below). Anyway, the importance here is to show that the anti-HTTP Response Splitting can be bypassed under some conditions.

Note that an HTTP (response) message containing multiple Content-Length headers is in violation of the HTTP/1.1 RFC [5].

Poisoning the cache of Apache 2.0.55 succeeded with multiple Content-Length headers were provided in the first HTTP response message (the injected header was the first one, of course).

Example response stream:

```
 HTTP/1.1 200 OK
 Termination-Token: cvb098srnwe23
 [...]
 Content-Length: 1234  <-- Injected header (first injection
                           point)
 [...]
 Content-Length: 5678
 [...]
 Termination-Header: cvb098srnwe23

 [... HTML data from the original response, 1234 bytes ...]
 HTTP/1.1 200 OK  <-- Injected complete HTTP response (second
                      injection point)
 Termination-Token: gotcha
 Content-Length: 46
 Termination-Header: gotcha

 <html>I can still do response splitting</html>  <-- End of
                                                     second
                                                     injection
 [... more HTML data from the original response ...]
```

Technique #3 - The PHP way - close, but no cigar
==

An impressive fine grained mechanism that attempts to prevent HTTP header injection, with HTTP Response Splitting as a special case ([7], [8], [9]), is implemented in the latest versions of PHP (5.1.2 and 4.4.2). The code in /main/SAPI.c (function sapi_header_op) performs the following:

```
1. Removal of the trailing sequence of CRs, LFs, SPs, and HTs,
   if such sequence exists.

2. Aborting if any LF found is not followed by SP or HT.

This really looks fine, except that Sun Java System Web Proxy
Server 4.0 happily accepts CR as an end of line marker. This
means that this proxy server can be exploited using CR only (no
LF whatsoever), so this anti- HTTP Response Splitting is not full
proof. Quite likely several other proxy servers are that liberal,
although strictly speaking, an HTTP message that has CR as an end
of line marker instead of CRLF is in violation of the RFC.
Using CR-only response, and a successful cache poisoning with CR-
only was demonstrated.

Handling additional patterns
============================

[4] suggests the following additional patterns for detecting HTTP
Response Splitting (on top of CRLF):

 <html

 <meta

 http/1.

Now, "<html" and "<meta" are located in the body of the injected
2nd response. Therefore, they can be easily hidden using UTF-7
encoding tricks [1] or UTF-16 encoding tricks, compression and
chunked-encoding [11]. Moreover, a malicious payload doesn't have
to use any of these. It suffices for most purposes to have a
payload such as:

 <script>...</script>

Or

 <script src=...></script>

Both IE 6.0 SP2 and Firefox 1.5 parse the <script> tag and
execute its code even if it is not nested inside an <html> tag.

As for the "http/1." pattern - some proxy servers are willing to
```

accept slight deviations from this pattern. For example, for Sun Java System Web Proxy Server 4.0 and DeleGate 8.11.5, "HTTP/" is enough for the response to be served nicely (and cached). So in their case, "HTTP/0.9", "HTTP/2.0" and "HTTP/01.0" can all be used successfully. In DeleGate's case, it's even possible to use "HTTP/ 1.0" (the Sun proxy server will not cache it - it probably needs an alphanumeric character after the forward slash).

Thus, Sun Java System Web Proxy Server 4.0 can be poisoned without using CRLF (i.e. using LF only) and without using the string "HTTP/1." (and instead, using "HTTP/2.1") and "<html" and "<meta". This was indeed demonstrated. In fact, it's even better - Sun Java System Web Proxy Server 4.0 will convert the response into a valid HTTP/1.1 response (i.e. convert the first line into "HTTP/1.1 ...").

Even if a proxy server won't cache the response if it is doesn't begin with HTTP/1.0 or HTTP/1.1, it may still treat the response as an HTTP/0.9 response [10] and send it back to the client (e.g. Apache 2.0.55, Sun Java System Web Proxy Server 4.0). Of course, it would have to wait until the connection is closed though, as there's no other way for the web server to inform the client of the end of the response message. In such case, the content is unlikely to be cached, but still, other tricks from [1] (such as cross site scripting, sending malicious data to an arbitrary client or receiving pages destined for another client) may be possible.

As for the major browsers - IE 6.0 SP2 will parse and cache responses starting with "HTTP/2.0", "HTTP/0.9" and "HTTP/01.1", and Mozilla Firefox 1.5 will parse and cache these, as well as "HTTP/foobar" and "HTTP /1.0".

Native double injection
=======================

On some application servers (e.g. IIS ASP), a redirection results in a 3xx response with a Location header containing the URL to redirect to, and an HTML body containing a reference to this same URL. In this case, a double injection is trivial. Yet one should keep in mind that the data injected is identical. An HTTP/0.9 response may be the only way to get around this, i.e. the injection may be:

```
Content-Length: N
Foo: <script>...</script>
```

Where N should be calculated so that it terminates the first response's body just after the "Foo:" injected in the body (the second injection).

Therefore, on such servers, double injection (needed for technique #2) is natively available.

Alternately, it is possible to inject data (without double CRLF, in order not to interfere with technique #2) such that at the second embedding point, it will be parsed as a partial HTTP response (incomplete header section). In such case, a double CRLF at the body of the web-server's first response, or the double CRLF in the web-server's second response (which terminates the second response's header section) would terminate this HTTP response. Consider the following injection text:

```
Content-Length: N
Foo: HTTP/1.1 200 OK
Content-Length: 0
Content-Type: text/html
Last-Modified: ...
Refresh: 0; URL=[http://www.evil.site/](http://www.evil.site/)
Bar:
```

At the first injection point, this will be interpreted (by a proxy server that uses the first Content-Length header) as an HTTP response whose size is N. N should be calculated such that it will position the proxy server right the string "Foo: " of the second injection. The proxy will therefore read the second injection and wait for the terminating double CRLF, which will complete the response and make it cacheable.

Of course, this method may fail if the HTML page contains a CRLF followed by data which the proxy cannot accept as an HTTP response header.

If the 2 latter alternatives (i.e. using HTTP/1.x response) are used, a problem arises: the injection string cannot directly use the two headers (Termination-Token and Termination-Header response) because it will flag the first response as invalid, and thus may alter the processing of the rest of the data. But if the double CRLF is provided someplace in the body of the first server response, then evading [6] can still succeed. It is assumed that a response from a non-compliant web server (i.e. a response that does not contain any one of the Termination-Token and Termination-Header response headers) should be accepted as valid by [6]. Otherwise, [6] would be impractical due to the majority of web-servers being non-compliant. Therefore, simply not providing those headers in the second (spoofed) response header section should evade [6].

On the other hand, if the terminating double CRLF is the one provided by the server as the termination sequence of the second response's header section, evading [6] will fail. After all, we assume that the web server complies with [6], and therefore each response header section would contain the two server generated headers (Termination-Token and Termination-Header). Hence, if the second injection relies on the second response header section to provide the double CRLF, it will fail to evade [6] due to the existence of those 2 server generated headers later in the second response headers. This method (of relying on the termination of the second server response header section), while not being a workaround for [6], may serve as a good counter-example to some other anti- HTTP Response Splitting ideas.

Detecting HTTP Response Splitting/Smuggling using the browser
==

As suggested in [12], a browser can be used in many cases to easily determine if a specific parameter in a specific script is vulnerable to HTTP Response Splitting. With the introduction of the above "anti HTTP Response Splitting" methods, the method presented in [12] may not work. However, as will be shown below, the same techniques employed above can also be used to verify the existence of HTTP Response Splitting/Smuggling.

Header injection at large (needed for technique #2) can still be verified the way [12] subscribes, i.e. injecting

```
%0d%0aSet-Cookie:%20HTTP_response_splitting%3dYES%0d%0aFoo:%20bar
```

Injecting LF only headers (technique #1) is a simple matter of trying the following string, and would succeed in both Microsoft IE 6.0 SP2 and Mozilla Firefox 1.5:

```
%0aSet-Cookie:%20HTTP_response_splitting%3dYES%0aFoo:%20bar
```

Injecting CR only headers (technique #3) is slightly more problematic. Both IE and Firefox in general do not parse CR as an end of header line. However, in some headers (probably those less critical for understanding the HTTP stream), IE is willing to accept CR as an end of headers. Fortunately, Location and Set-Cookie are such headers. Therefore, using IE 6.0 SP2 (but not Firefox, unfortunately), we can use the following string to verify:

```
%0dSet-Cookie:%20HTTP_response_splitting%3dYES%0dFoo:%20bar
```

Recommendations
===============

Providers of anti- HTTP response splitting solutions

Do not rely on double CRLF patterns, on the existence of even a single CRLF, or on existence of the "HTTP/1." string or of HTML tags. Do not assume that HTTP Response Splitting requires "breaking out" of the header section (consult the above counter-examples to make sure that the solution indeed covers at least those scenarios). Instead, focus on what is by definition invalid in HTTP responses, such as CRs and LFs in header names and values (as recommended in [1]). When doing so, keep in mind that the logical form of the data (the characters CR and LF) may be represented in various ways as physical characters (e.g. raw characters, URL-encoded, the IIS-specific %uHHHH encoding, UTF-8 overlong/invalid encoding, etc.), and may be delivered not only in the URL, but also in the body parameters and possibly in other locations as well (e.g. headers and path). Simply blocking dangerous characters in the URL, in their raw form and their URL-encoded form (as hinted in [3]) is insufficient.

HTTP client vendors (including browsers and proxy servers)

Disallow invalid or ambiguous responses such as discussed above (but do not limit this treatment to those examples, it's very likely that there are more such problems). Ideally, convert such response to an error response (perhaps with 5xx status code) and terminate the TCP connection.
Also, consider the detection method described in [13].

Security testers

Pay heed to the extension of the detection method in [12] to HTTP Response Smuggling, and learn the patterns suggested above.

Summary
=======

HTTP Response Smuggling is possible because some of the protection mechanisms suggested address symptoms, not root cause, of the injection into the HTTP response headers problem. It also exploits the liberal and tolerant parsing exhibited by several proxy servers and. The net result is that the classic HTTP Response Splitting is still at large possible, requiring minimal modifications to overcome the current protection mechanism.

Additional research directions
==============================

While HTTP Response Smuggling was developed to bypass several anti- HTTP Response Splitting mechanisms, no doubt it has many other applications. One such direction is bypassing content filtering. A malicious web server can send HTTP responses that may be interpreted in one manner (as innocent responses) by a content filtering gateway, and in another manner completely (as malicious pages) by the end client (browser). Another direction is spoofed indexing, wherein a search engine parses the data stream one way, while the actual clients may parse it differently. Likewise, it may be possible to generalize and serve different content to different clients, based on difference in the way clients parse the HTTP response stream. Finally, it should be noted that there are many more response smuggling techniques, of which only 3 were discussed in the paper due to brevity and clarity considerations. Such techniques can be explored to exploit other scenarios (combinations of servers, proxy servers, browsers and protection techniques).

A note about terminology
========================

This paper concludes the HTTP {Request,Response} x {Splitting,Smuggling} quartet. Since the first work was introduced ([1], almost 2 years ago), there was some misunderstanding of the terms and concepts in the various works

(e.g. the difference between [1] and [2]). In order to clarify
the terminology, here are two definitions:

Splitting - the act of forcing a sender of (HTTP) messages to
emit data stream consisting of more messages than the sender's
intension. The messages sent are 100% valid and RFC compliant.

Smuggling - the act of forcing a sender of (HTTP) messages to
emit data stream which may be parsed as a different set of
messages (i.e. dislocated message boundaries) than the sender's
intention. This is done by virtue of forcing the sender to emit
non-standard messages which can be interpreted in more than one
way.

Both terms (when applied to HTTP requests/responses) belong to
the peripheral web security world, as described in [14].

References
==========

[1] "Divide and Conquer - HTTP Response Splitting, Web Cache
Poisoning Attacks, and Other Topics", Amit Klein, March 2004
[http://www.packetstormsecurity.org/papers/general/whitepaper_httpresponse.pdf](http://www.packetstormsecurity.org/pap

[2] "HTTP Request Smuggling", Chaim Linhart, Amit Klein, Ronen
Heled, Steve Orrin, June 2005
[http://www.watchfire.com/resources/HTTP-Request-Smuggling.pdf](http://www.watchfire.com/resources/HTTP-Request-Smug

[3] "Blocking HTTP Attacks Using CPL", BlueCoat Technical Brief
[http://www.bluecoat.com/downloads/support/tb_blocking_HTTP_attacks.pdf](http://www.bluecoat.com/downloads/support/

[4] "Learn How To Configure Your ISA 2004 Server To Block HTTP
Response Splitting Attacks", Microsoft document
[http://download.microsoft.com/download/a/6/0/a609cd64-1f53-4c0a-b8d7-](http://download.microsoft.com/download/a/6/
b38be65fa1c5/http%20_response_splitting_attacks.doc

[5] "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June
1999
http://www.ietf.org/rfc/rfc2616.txt

[6] "Effective Countermeasure to HTTP Response Splitting", Aaron
Emigh, anti-fraud () lists cacert org mailing list submission,
September 11th, 2005.
http://www.mail-archive.com/anti-fraud () lists cacert org/msg00121.html

[7] "PHP 5.1.2. Release Announcement", PHP website
http://www.php.net/release_5_1_2.php

[8] "PHP 4.4.2. Release Announcement", PHP website
http://www.php.net/release_4_4_2.php

[9] "Goodbye HTTP Response Splitting, and thanks for all the
fish", Stefan Esser, "PHP Security Blog" blog post, January 12th,
2006
[http://blog.php-security.org/archives/28-Goodbye-HTTP-Response-Splitting,-and-thanks-for-](http://blog.php-security.org/arc
all-the-fish.html

[10] "The Original HTTP As Defined in 1991", Tim Berners-Lee,
1991
http://www.w3.org/Protocols/HTTP/AsImplemented.html

[11] "Bypassing content filtering whitepaper", 3APA3A
http://www.security.nnov.ru/advisories/content.asp

[12] "Detecting and Testing HTTP Response Splitting Using a
Browser", Amit Klein, WebAppSem mailing list submission, October
14th, 2004
http://www.securityfocus.com/archive/107/378523

[13] "Detecting and Preventing HTTP Response Splitting and HTTP
Request Smuggling Attacks at the TCP Level", Amit Klein, BugTraq
mailing list submission, August 15th, 2005
http://www.securityfocus.com/archive/1/408135

[14] "Meanwhile, on the other side of the web server", Amit
Klein, June 10th, 2005
http://www.itsecurity.com/security.htm?s=3957

Appendix
========

Web Cache Poisoning with Sun Java System Web Proxy Server 4.0
===

Here are some practical considerations to be taken into account
when poisoning the cache of Sun Java System Web Proxy Server 4.0
(B05/10/2005) via HTTP Response Splitting (or Smuggling).

1. The Sun proxy server has some kind of buffering or packet-boundary parsing of the HTTP response stream. Therefore, padding of few thousand bytes is required between the end of the first response and the beginning of the second response. In the author's experience, 3000-6000 bytes usually suffice.

2. The Sun proxy server has a unique parsing mechanism wherein it scans for the first response line ("HTTP/..."), so the exact position of the response is less critical (compared to the precision required to poison other cache servers).

3. Due to some timing issues, it's much easier to poison Sun proxy server's cache with a (second) HTTP message whose Content-Length is 0. This is still interesting because a redirection can be forced (e.g. via a Refresh header). That is, it's possible to poison the cache with a fake 0 length homepage of the target website, refreshing itself immediately to the attacker's website (classic defacement). Poisoning the cache with 0-length header has high rate of success (>50%), while for non-empty response, it's lower (though was demonstrated several times). It seems that the reason is that Sun proxy server terminates a 0-length response right after the headers, regardless of what's following. When facing a non-empty response, it will not cache the response if superfluous data exists. This means that in order to successfully poison the cache with non-empty response, the real second response from the web server should be taken into account, and even then, there are some timing issues.

4. It seems that Sun proxy server will cache all URLs except root resources (e.g. [http:*//www.some.site/](http://www.some.site/)).*

5. Forcing a cache revalidation is done using the "Pragma: no-cache" HTTP request header. This header should therefore be included with the second request (the one made for the poisoned resource).

Current thread:

- **Whitepaper by Amit Klein: "HTTP Response Smuggling"** *Amit Klein (AKsecurity) (Feb 21)*