Jedi Mind Tricks: HTTP Request Smuggling

Lori MacVittie, 2009-23-04

How to defeat the ancient Jedi mind trick known as HTTP Request Smuggling.

HTTP Request Smuggling (HRS) is not a new technique; it's been around since 2005. It takes advantage of architectures where one or more intermediaries (proxies) are deployed between the client and the server. HRS is can be used to poison web-caches and bypass security solutions such as web application firewalls as well as for the delivery of malicious payloads such as worms, viruses, and those used to exploit known vulnerabilities in web and application servers.

These are not the requests you are looking for.

The good news is that to exploit HRS, according to OWASP, "some specific conditions must exist, such as the presence of specific proxy system and version such as SunOne Proxy 3.6 (SP4) or FW-1/FP4-R55W beta or an XSS vulnerability in the web server." The "in the web server" should more correctly read "in a page or application hosted on the web server", as it is through a traditional XSS vulnerability in a web page/application that such attacks can be carried out if they are passed on by an intermediary.

The bad news is that the nature of cloud computing and virtualized architectures makes an attack based on HRS more likely to find "some specific conditions" than when it was first discovered. This is because intermediaries (proxies) are a fact of life in cloud and virtualized architectures. The variety of intermediaries used - both commercial and custom-built - to architect such environments increases the possibility that one or more of them will be vulnerable to HRS. And as Jeremiah Grossman tirelessly points out 82% of websites have at least one security flaw, and XSS is high in probability to be that one flaw.

The likelihood of being vulnerable to HRS is increased by the use of HTTP pipelining and without careful consideration and inspection of HTTP headers and payloads even security devices designed to prevent web application attacks may be vulnerable themselves.

THESE ARE NOT THE REQUESTS YOU ARE LOOKING FOR

HRS works by exploiting the way in which HTTP endpoints parse and interpret the protocol and counts on the lax enforcement of the HTTP specification (RFC 2616). There are several ways in which the protocol is exploited by HRS, the most prevalent of which is to ignore section 14.13 in which it clearly states (if you're a reader of BNF, at least) there should be ***one*** and only one ***Content-Length*** header. Also commonly shown in examples of HRS is the exploitation of the ability to define custom HTTP headers.

What HRS does is trick an endpoint into thinking it is handling one request - a safe one - while smuggling a malicious request inside the safe one that will be parsed and executed by the application server. It's like using a Jedi mind-trick to convince the proxy to just ignore the droids, er, one of the requests.

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 behavior of an endpoint when receiving multiple HTTP headers and partly because endpoints have always been more forgiving of clients that take liberties with the HTTP protocol than they should be. So some endpoints ignore the first, or the second, and then use the data included in the Content-Length to parse the request. This can be used to direct proxies to treat requests as data and vice-versa, which can confuse endpoints and trick them into executing malicious requests hidden inside legitimate requests.

The potential for exploitation is actually exacerbated by the use of HTTP pipelining, as some clients piggyback multiple HTTP requests in a single packet. Those requests can include full HTTP headers, including Content-Length. This means that it is possible to hide a smuggled request deep inside pipelined requests and can increase the possibility of an HTTP Response Splitting attack. Interestingly enough, Jeremiah Grossman notes HTTP response splitting vulnerabilities in web applications among the top ten vulnerabilities discovered by White Hat Sentinel.

Conditions are rapidly becoming perfect for these two types of attacks to bear fruit.

HOW TO AVOID THE EFFECTS OF JEDI MIND TRICKS

If you aren't Jabba the Hut then you either need smarter Storm Troopers or solutions with an ability to recognize - and stop - such mind tricks.

Intermediaries should enforce the HTTP protocol - specifically the presence of only ONE Content-Length header. At a minimum, the outermost (edge) intermediary should validate that HTTP requests have only one content-length header.

1. **Network-side scripting.** If you're using an application delivery controller as the edge intermediary (a common enough architecture) you can take advantage of network-side scripting to inspect requests and enforce the 1:1 Content-Length header to request rule.

   Example using iRules (thanks to uber DevCentral member hoolio):

   ```
   when HTTP_REQUEST {

       # Check if there is more than one Content-Length header
       if {[HTTP::header count "Content-Length"] > 1}{

          # Reset the connection
          reject

          # Stop processing this event in this iRule
          return
       }

       # If the rule is still running, check if the Content-Length header exists
       # and has a value less than or equal to 0
       if {[HTTP::header exists "Content-Length"] && [HTTP::header value Content-Length"] <= 0}{

          # Reset the connection
          reject
       }
    }
   ```

   mod_security and mod_rewrite also provide network-side scripting capabilities that can be used to implement this functionality, as do most other application delivery controllers on the market today.

   Also examine the XSS mitigation capabilities and compliance with the HTTP specification of other intermediaries such as web application firewalls.

2. **Protocol security solutions.** Some intermediaries have the capability of strictly enforcing application protocols such as FTP, SMTP, and HTTP. If not included in the base solution, check to see if this is an add-on option.
3. **Secure coding.** Obviously if your application is not vulnerable to XSS attacks, this mitigates one of the potential attack vectors of this exploit. Ensure you are validating input and all requests, and if possible validate the HTTP request for compliance with RFC 2616 in your application.
4. **Reconsider the use of HTTP pipelining.** While I'm still tossed up regarding the performance benefits, others are not and are enthusiastic about its use to improve page load times. Given that HTTP pipelining can increase the

not and are enthusiastic about its use to improve page load times. Given that HTTP pipelining can increase the possibility of a successful HTTP Request Smuggling/Response Splitting attack, you may want to reconsider its use and find alternative methods of improving page response time. A thorough risk/benefit analysis is in order when considering the best way to improve performance without also increasing your vulnerability to attack.

5. **Evaluate each intermediary.** Re-evaluate each intermediary in the chain of proxies between the client and your server for possible vulnerabilities to HRS. Investigate whether there are simple options you can enable or disable in the intermediary to tighten enforcement of the HTTP protocol and minimize the risk of being exploited.

And in case you wondering: *HAN SHOT FIRST.*

- SecuriTeam - HTTP Request Smuggling
- HTTP pipelining - indirect security implications
- CGI Security: HTTP Request Smuggling
- I am in your HTTP headers, attacking your application
- Stop brute force listing of HTTP OPTIONS with network-side scripting
- What's the difference between a web application and a blog?
- 3 reasons you need a WAF even though your code is (you think) secure
- The Concise Guide to Proxies
- HTTP Request Smuggling - OWASP
- Understanding network-side scripting