iRule Security 101 - #08 - Limiting POST Data

Joe Pruitt, 2008-17-01

With the increasing popularity of the internet for application delivery, user supplied information is almost a given. The method of choice for uploaded content is HTML forms making use of the POST HTTP command. One can craft HTML forms to limit the length of input form fields, but there are tools available to "sniff" this post data and generate requests that look for exploits in POST data limits. This article will describe how to interrogate a HTTP POST request and reject requests containing large HTTP POST data lengths. Other articles in the series:

- iRule Security 101 – #1 – HTTP Version
- iRule Security 101 – #02 – HTTP Methods and Cross Site Tracing
- iRule Security 101 – #03 – HTML Comments
- iRule Security 101 – #04 – Masking Application Platform
- iRule Security 101 – #05 – Avoiding Path Traversal
- iRule Security 101 – #06 – HTTP Referer
- iRule Security 101 – #07 – FTP Proxy
- iRule Security 101 – #08 – Limiting POST Data
- iRule Security 101 – #09 – Command Execution

Background

For those non-dev folks out there, whenever you type something into an edit box on a web page and click a "submit" type button, in most cases whats going on under the seems is a HTTP request is made to the target server with the data packaged in a HTTP POST request. An example of a HTML form that sends a single value named "hostname" to a web application with a HTTP POST command.

```
<form name="loginForm" action="http://somesite.com/loginform.html" method="POST">
  UserName: <input type="text" name="username" maxlength="20"/> <input type="submit" value="Submit"/>
</form>
```

The generated HTTP request will look something like this if the user typed "Joe" into the username text box:

```
POST /loginform.html HTTP/1.1
Host: somesite.com
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)
Content-Type: application/x-www-form-urlencoded
Content-Length: 12

username=Joe
```

As you see, the POST Data contains the name=value pairs for all of the form elements.

The security issue

Let's assume the application processing this request has assumed that since it's html front end limited the text to 20 characters (as specified by the maxlength attribute in the input element) it only had to test for entries limited to 20 characters. Unit tests were run and passed and the application was put live. Here comes the potential problem...

Let's say hacker Fred comes in and thinks to himself. Hmmm. I wonder what happens when if I send this application a multi-megabyte string for the username input value.

Fred packages up his own request like this

```
POST /loginform.html HTTP/1.1
Host: somesite.com
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)
Content-Type: application/x-www-form-urlencoded
Content-Length: 1000009
username=aaa...aaa
```

Where "aaa...aaa" is 1000000 characters long and sends it to the application with the tcp client of his choice. Since the server application didn't test for large string sizes, odds are a memory error could occur such as a buffer exploit, memory corruption or something more extreme. Bad things can happen such as server crashes and possibly data corruption and leakage of sensitive information.

The solution

iRules provide a very easy way to inspect the Content-Length of a HTTP request and block any requests that violate length constraints before the request even makes it to the application server. The following iRule will select an arbitrary limit of 1024 characters for the HTTP POST Data.

```
when RULE_INIT {
  set DEBUG 1
  set sec_http_max_post_data_length 1024
}

when HTTP_REQUEST {
  if { [HTTP::method] equals "POST" } {
    set len [HTTP::header "Content-Length"]
      if { [expr $len > $::sec_http_max_post_data_length] } {
        log local0. "  SEC-ALERT: POST Length: uri=[HTTP::uri]; len=$len; max_len=$::sec_http_max_pos
        reject
      }
    }
  }
}
```

If the Content-Length is larger than this limit, a message is sent to the system log and the connection is rejected. The application will never see the request and you've just avoided a buffer exploit attempt.

Conclusion

This is not a end-all-be-all solution for form protection. In fact, there are entire products developed to perform deep field inspection for HTML forms. But this solution does allow for an arbitrary first line of defense, or a quick way to protect a newly discovered exploit until an application fix is made.

Get the Flash Player to see this player.