iRule Security 101 - #04 - Masking Application Platform

Joe Pruitt, 2007-07-09

In this session of iRules Security 101, I'll show you how to hide your backend server application platform from the outside public. When you are browsing a website, you are likely to look at the URI's to determine what platform the application is running on. For instance, if you saw "http://www.foo.com/foo.asp", you would likely assume that the backend application was an ASP application running under Microsoft IIS. Likewise, if you saw "http://www.foo.com/foo.jsp", you would assume it was a java server platform. Well, in the world of exploits, wouldn't be best to hide the platform from application is running on from the outside world? This article will illustrate one way to do so. Other articles in the series:

In this article, we will look at the URI extensions that your application uses. For this example, let's assume that your backend webserver is running an ASP application and you don't want the world to know it. We'll create a new "public" extension that the outside world will interact. In this example, we'll use ".joe". Hey, I wrote the article, so I get to name the extension B-).

Let's first look at the HTTP request:

```
when HTTP_REQUEST {
  # Don't allow data to be chunked. This ensures we don't get
  # a comment that is spread across two chunked boundaries.
  if { [HTTP::version] eq "1.1" } {
    if { [HTTP::header is_keepalive] } { HTTP::header replace "Connection" "Keep-Alive" }
    HTTP::version "1.0"
  }

  set orig_uri [HTTP::uri]
  log local0. "Old URI: $orig_uri"
  switch -glob $orig_uri {
    "*.html*" {
      log local0. "Found request to internal resource.  Redirect to external resource"
      set new_uri [string map {".html" ".joe"} [HTTP::uri]]
      HTTP::redirect "http://[HTTP::host]$new_uri"
    }
    "*.jsp*" {
      log local0. "Found request to internal resource.  Redirect to external resource"
      set new_uri [string map {".jsp" ".joe"} [HTTP::uri]]
      HTTP::redirect "http://[HTTP::host]$new_uri"
```

```
    }
    "*.asp*" {
      log local0. "Found request to internal resource.  Redirect to external resource"
      set new_uri [string map {".asp" ".joe"} [HTTP::uri]]
      HTTP::redirect "http://[HTTP::host]$new_uri"
    }
    "*.joe*" {
      log local0. "Found external resource request, mapping URI to internal name"
      HTTP::uri [string map {".joe" ".asp"} [HTTP::uri]]
    }
  }
}
```

The HTTP_REQUEST event looks at the incoming URI and for the some common application extensions (.html, .jsp, and .asp), we'll redirect to the public facing ".joe" uri. The reason I did this if we only masked the internal application type of .asp, then all other requests wouuld return 404 (file not founds) from the server and the .asp requests would redirect. This would be a good sign we are hiding the .asp extension. By redirectly multiple extensions, there is no indication what the true extension really is. Here are some example redirects that this would generate:

```
http://www.foo.com/index.html -> http://www.foo.com/index.joe
http://www.foo.com/default.asp -> http://www.foo.com/default.joe
http://www.foo.com/login.jsp -> http://www.foo.com/login.joe
```

The last case in the switch statement turns all .joe requests to the true .asp requests before sending passing the request to the application server. Here are some sample URL transformations that would be made

```
http://www.foo.com/default.joe --> http://www.foo.com/default.asp
http://www.foo.com/login.joe?username=foobar --> http://www.foo.com/login.asp?username=foobar
```

Now that the request is taken care of, we most likely should modify the responses to change all embedded URLs in the response content so that they reflect the external extension.

```
when HTTP_RESPONSE {
  if { $orig_uri ends_with ".joe" } {
    # Ensure all of the HTTP response is collected
    if { [HTTP::header exists "Content-Length"] } {
       set content_length [HTTP::header "Content-Length"]
    } else {
       set content_length 1000000
    }
    if { $content_length > 0 } {
       HTTP::collect $content_length
    }
  }
  if { [HTTP::header exists "Server" ] } {
    HTTP::header replace "Server" "Joe's Awesome App Server"
  }
}

when HTTP_RESPONSE_DATA {
  set new_payload [string map {".asp" ".joe"} [HTTP::payload]]
  HTTP::payload replace 0 [HTTP::payload length] $new_payload
}
```

In the HTTP_RESPONSE event, we check to see if the original uri was to our external extension. If so, trigger a collection of the payload from the backend server and perform a simple string replacement in the HTTP_RESPONSE_DATA event and update the response payload with the "HTTP::payload replace" command (make sure you have rechunking enabled in your HTTP profile). Oh, and make sure you check out the last line in the HTTP_RESPONSE event. Another giveaway as to your server type, is the "Server" HTTP response header. I went ahead and modified this to remove any server based identification.

There are several ways you could enhance this by adding support for default index pages as well as adding support for other response content coming from non ".joe" based uri requests. You may also have more than one internal extension that you want to hide (.dll clearly indicates a windows machine). One could create multiple mappings (.joe-a -> .asp, .joe-b -> .dll, .joe-c -> .cgi, ...) and make the approprate redirections in the request and modifications in the response.

Keep in mind that there are many ways a server can identify itself to the outside world. This iRule doesn't protect against all types of server "signatures" but gives you a good start.

F5 Networks, Inc. | 401 Elliot Avenue West, Seattle, WA 98119 | 888-882-4447 | f5.com

F5 Networks, Inc. Corporate Headquarters info@f5.com	F5 Networks Asia-Pacific apacinfo@f5.com	F5 Networks Ltd. Europe/Middle-East/Africa emeainfo@f5.com	F5 Networks Japan K.K. f5j-info@f5.com