Security Irules 101: You can’t always get what you want.. or can you?

Josh Michaels, 2014-08-11

Introduction

iRules are a powerful tool in the F5 administrators arsenal. They allow administrators to adapt and customize the F5 to their needs. They provide extensive power for security engineers as well. We've decided it's time to revisit the Security iRules 101, with updated content, and 100% more monkeys!

In section 2 of the series, we are going to talk about controlling our HTTP. Browser based web communication use the power of RFC 2616 (this plus many others) to tell browsers how to get data and what to do with it.

A browser uses a defined set of methods to communicate its needs to the servers. The basic set of methods, defined in RFC 2616.9.0 are (in basic humanese):

> **OPTIONS:** Request the server tell you what methods are allowed, as well as other optional features implemented on the server
>
> **GET:** Client is asking the server to return a resource/information.
>
> **HEAD:** Just give the client the headers, no message body.
>
> **POST:** Sending data into the server that you expect it to do something with. Posting a message, logging in, etc.
>
> **PUT:** Take what the client sends, and store it. Example here is old school file transfer.
>
> **DELETE:** Please delete what's at the URI. Pretty please? There is not guarantee that the deletion is honored.
>
> **TRACE:** Trace my request all the way back to the origin server, and tell me where it went. Not a good thing to allow, as it gives out path information. (read: don't let that be used publically.. ever)
>
> **CONNECT:** Reserved by the original RFC to use with dynamic tunneling.

Is that all? Heck no. Check out: Anne's Blog for a list of some of the lesser known methods.

Why do we care?

So why do we care about what methods are allowed into a server? If the application team hasn't taken due diligence with the app (which never happens of course) or the server team just slapped a webserver up and haven't gotten around to finishing that last bit of configuration.

Do we sit around and wait for things to get fixed? Hell no, we charge on!

Roll in the iRules

We know that iRules can read and effect near every bit and byte that comes across the F5, so what can we do for HTTP method limiting?

The amount of built-in HTTP commands in iRules is pretty fantabulous. Pretty much every portion of the HTTP payload is accessible, as seen in the HTTP Wiki on Devcentral. For the matter at hand, we

the HTTP payload is accessible, as seen in the HTTP Wiki on DevCentral. For the matter at hand, we are looking towards HTTP::method. It simply returns the method from the HTTP request.

```
when HTTP_REQUEST { log local0. "HTTP Method: [HTTP::method]" }
```

We could just build a one off iRule to block based on methods hard coded into the rule. That means that anytime we need to add or remove a method from the rule, we have to edit the rule. Meh, I hate having to change variables in the rules, so let's think. What can an iRule access on the F5, that stores values, and is changeable via the GUI? You got it, it's data groups time .

The rule:

```
when HTTP_REQUEST {
    if { [class match [string tolower [HTTP::method]] contains disallowed_methods] }
{
        log local0. "BAD METHOD!!! [HTTP::method] -"
        }
```

The datagroup:

```
ltm data-group internal disallowed_methods {
    records {
        post { }
    }
    type string
}
```

So, here we just log posts to the webserver locally to the F5. Using a *Class Match* command we compare a lower case version of the method versus the entries in the datagroup *disallowed_methods.* It's good, but we can do better, with a positive security model.

```
ltm data-group internal allowed_http {
    records {
        get { }
        post { }
    }
    type string
}
```

```
ltm data-group internal allowed_nonhttp {
    records {
        copy { }
        lock { }
        mkcol { }
        move { }
        propfind { }
        proppatch { }
        unlock { }
    }
    type string
}
```

```
when RULE_INIT {
set static::Servers {
<html>
<head>
</head>
<body>
<p>Disallowed HTTP Method</p>
</body>
</html>
}
}
when HTTP_REQUEST {
   if { [class match [string tolower [HTTP::method]] contains allowed_nonhttp] } {
      log local0. "Good, but no http!! -- [HTTP::method] --"
      HTTP::disable
      return
   }
   if { [class match [string tolower [HTTP::method]] contains allowed_http] } {
      log local0. "Good request [HTTP::method]"
      return
      }
   HTTP::respond 405 content [subst $static::Servers]
}
```

So in this Rule, we create a Response page (set Servers variable). Then, on every HTTP_REQUEST, we look for our allowed_nonhttp methods, then look for allowed_http methods. If we don't match either of them, respond back with the 405 "Method Not allowed" page, from the F5 Platform.

How is it a positive security model? Well, it defines what we want to allow through, and everything else, gets dropped to the 405. Why is this nice? Because we can just define the methods we want to allow, not each method we don't want. It makes it simpler to manage, as new methods come out, we don't have to update the datagroups.

Conclusion

A quick iRule that improves the security posture of a web property. We used the power of the data groups, quick HTTP rip aparts, and the ability to return web pages from the platform directly, to quickly solve any METHODS issues.

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