Translating a Non-secure Web Address to a Secure one, Version 1.1

By R. G. Sparber

Protected by Creative Commons.[1]

Scope

The code presented here only applies to Linux servers. It has only been used with the GoDaddy hosting who supplied it to me as a template. I decided to write this explanation because all of the tutorials I found assume more background than I initially possessed.

The problem that I needed to solve was being able to accept requests for my old web site which was not secure (`http://`) and translate them to that same name but secure (`https://`).

History

When I first set up my web site, I chose `sparber.org` as my domain. Then I defined the sub-domain `rick` for all of my files. The result was the address `sparber.org/rick` which can also be written `rick.sparber.org`. The intent was that other members of my family could have their own sub-domains. For example, Sally could be defined as a sub-domain which would create the address `sally.sparber.org`. Over the years it became evident that they weren't interested in populating their sub-domains so this feature was never used.

Recently there has been a push by GoDaddy to get all web sites more secure. This is done by buying one or more Secure Sockets Layer (SSL) Certificates. These certificates are checked by the user's browser and provide a level of assurance that it is safe to proceed. Buying one certificate cost around $70 per year. It would protect `sparber.org` but not `rick.sparber.org`. To do that, I would need to buy the next larger package which includes five certificate and costs around $140 per year. Since I gain no revenue from my web site, this cost was

[1] This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

getting painful. At first, I moved all of my files out of my sub-domain so users could access them via `sparber.org` rather than `rick.sparber.org`. Then I only need one certificate. My understanding was that I could detect `rick.sparber.org` requests and redirect them to `sparber.org`. But then I discovered a “Catch-22”s.

When a user entered the old address, they were warned that it was not secure. At that point, it would be prudent for the user to not proceed. I did try many different methods to automatically perform this redirect but none worked. Four full days were wasted on that effort.

I often called GoDaddy staff for help. They were always friendly and attentive. However, after a while I discovered they were not of equal skill level. Ultimately, I reached someone with the perfect solution: rename `sparber.org` to `rick.sparber.org` and have only one SSL Certificate. He had me download a template which I had tried dozens of time before. But this time, GoDaddy staff spent a full 1 ½ hours making changes in the server including a restart. Then, magically, it all worked. To put it mildly, the code in this template was necessary but not sufficient.

Before I lose what I learned about this code, I thought I would document it. Someone out there will be following in my footsteps.

The Template Filled Out

The following 4 lines of code is saved in a `.htaccess` file. This file is placed at the “root”. The root is where all accesses to the web site start and is called `public_html`. The server looks here for an `index.html` file for what to display as the home page. However, if there is a `.htaccess` file, it looks there first. This file has a large number of possible uses, one of which is to forward addresses.

```
RewriteEngine On
RewriteCond %{SERVER_PORT} 80
RewriteCond %{HTTP_HOST} ^(www\.)?rick\.sparber\.org
RewriteRule ^(.*)$ https\:\/\/rick\.sparber\.org\/$1 [R,L]
```

I will now take this apart and, to the best of my knowledge, explain what is going on.

```
RewriteEngine On
RewriteCond %{SERVER_PORT} 80
RewriteCond %{HTTP_HOST} ^(www\.)?rick\.sparber\.org
RewriteRule ^(.*)$ https\:\/\/rick\.sparber\.org\/$1 [R,L]
```

`RewriteEngine On` tells the server we want to use the Rewrite functionality.

The next two lines define what conditions must be met before we change anything related to the user.

`RewriteCond %{SERVER_PORT} 80`
`RewriteCond` says that a condition is about to be defined
`%{SERVER_PORT}` consists of two parts. **`%{ }`** means that a system variable is going to specified. The system is the Linux server. The variable name is not case sensitive so **`server_port`** would work too.
`SERVER_PORT` is the system variable. I could not find anything about it but apparently it must equal **`80`** before we can proceed. If **`%{SERVER_PORT}`** does not equal **`80`**, the server stops processing the `.htaccess` file and moves on to the `index.html file.`

`RewriteCond %{HTTP_HOST} ^(www\.)?rick\.sparber\.org`
`RewriteCond` says that a condition is about to be defined
`%{HTTP_HOST}` This system variable contains the address. As best as I can figure, it does not contain `http://` if it was entered into your browser.
The value represented by **`%{HTTP_HOST}`** is compared to the remainder of this line: **`^(www\.)?rick\.sparber\.org`**
The "**^**" means that we start comparing text at the beginning of the string. So if **`%{HTTP_HOST}`** equaled `abcwww` it would not match. But `www` at the start would match.
`(www\.)?` has a lot going on. First of all, "**`( )`**" means we are defining a value which is part of the comparison. By adding the "**?**" at the end, we are saying that this value is optional. However, since it follows the "**^**", it must be at the start of the line. The value consists of the letters **`www`** followed by a period**.** The "\" tells the system that I really mean period. Without this "escape", it would read the period as a symbol meaning any single character. So `www.` could, for example, represent `wwwx` but `www\.` only represents "`www.`". Given **`(www\.)?`** we accept the case of the user entering `www.rick.sparber.org` or just `rick.sparber.org.`

`RewriteCond %{HTTP_HOST} ^(www\.)?rick\.sparber\.org` (continued)

Next we have **`rick\.sparber\.org`** which is the remainder of the string being compared to **`%{HTTP_HOST}`**. It uses two escapes so the system sees "`rick.sparber.org`".

Putting this all back together,
`RewriteCond %{HTTP_HOST} ^(www\.)?rick\.sparber\.org`

tells the system to take the contents of the system variable **`HTTP_HOST`** and compare it to "**`www.rick.sparber.org`**" and to "**`rick.sparber.org`**". If either of these match, we proceed. If neither match, we terminate and go looking for the `index.html` file.

```
RewriteEngine On
RewriteCond %{SERVER_PORT} 80
RewriteCond %{HTTP_HOST} ^(www\.)?rick\.sparber\.org
RewriteRule ^(.*)$ https\:\/\/rick\.sparber\.org\/$1 [R,L]
```

When we proceed, the last line of the code is executed:

`RewriteRule ^(.*)$ https\:\/\/rick\.sparber\.org\/$1 [R,L]`

This command is a Rule rather than a Condition. This rule is looking at *what has not yet been matched* in **`HTTP_HOST`** and can translate part or all of it. Apparently, it ignores the "/" that would exist when a user enters

`rick.sparber.org/<file name>.pdf`

`^(.*)$` Starts with a ^ which means to look at the remainder of **`HTTP_HOST`** starting at the first character. We again have "**`( )`**" which means we are defining a quantity. Inside of these parentheses we have "**.**" but there is no escape character in front of it. That means we are defining a single character which can be anything, including no character at all. Next we have "**`*`**" which means zero or more times. Therefore, it is saying that we are looking for a series of characters that starts at the beginning of what follows **`rick.sparber.org/`** and it can be of any length. The "**`$`**" says to look all the way to the end of the line. So if the user entered `rick.sparber.org/amm.pdf`, we would collect `amm.pdf`. I say "collect" because the code is doing more than just matching, it is saving this value. More on this later.

`RewriteRule ^(.*)$ https\:\/\/rick\.sparber\.org\/$1 [R,L]`
(continued)

Next we have **`https\:\/\/rick\.sparber\.org\/`** which is loaded with escapes. This means we want the string **`https://rick.sparber.org/`**. Ah, so this is where we redirect the address to the secure form of **`https`**. When the system see https, it starts looking for the corresponding SSL certificate. If found, it has the browser display “Secure” next to the address bar along with an icon that looks like a lock.

`$1` comes next. This means that we should take the string of characters captured by **`^(.*)$`** and put it right after **`https://rick.sparber.org/`**. For example, if the user entered `rick.sparber.org/abc/def.pdf`, this command would assemble `http://rick.sparber.org/abc/def.pdf`. If they entered `rick.sparber.org/def.pdf` this command would assemble `http://rick.sparber.org/def.pdf`. The key is that **`^(.*)$`** soaks up all that is left in the command line and saves it as **`$1`**. Note that if I had another **`( )`** construct before the end of the line, it would be called **`$2`**.

The last part of the command is the optional “Flag” section defined by **`[ ]`**. There are two flags present here: **`R`** and **`L`**. The **`R`** causes the string just built to be sent back to the user’s browser which is told this is a Redirect. The user then sees this new string in the browser’s address bar and, a moment later, the corresponding page below it. The **`L`** tells the server to stop executing the code. It then goes off to find the `index.html` file so it can build the page.

Acknowledgements

Thanks to the staff at GoDaddy for the many hours of assistance.

I welcome your comments and questions.

If you wish to be contacted each time I publish an article, email me with just "Article Alias" in the subject line.

Rick Sparber
Rgsparber.ha@gmail.com
Rick.Sparber.org