SSL+PHP Web Server

Steven Lu (sjlu@eden.rutgers.edu)

May 7, 2012

1 Introduction

The following assignment implements SSL and PHP into a deliverable web server. This was inspired by the lectures of IO, directories, Web servers, threads and SSL. Another major inspiration was that modern web servers can process dynamic data (or have scripts). In this case, I wanted to implement PHP or Pre-hypertext Processor execution into the server.

Since we implemented a very small web server that serves a single static file, the web server itself had to be implemented first, being able to handle the header request from the browser, process it, and deliver acceptable content to the client's browser. This also means that we have to also handle the response headers properly (such as 404 Not Found, 200 OK and 500 Internal Server Error).

The second major component was implementing SSL encryption into the server. To do so, we had to import certain C libraries such as GNU TLS in order to handle the encryption. This means that we had to detect the incoming SSL request, create a handshake with the browser, encrypt the data based on the session and then finally deliver the data through a secure pipe.

The last major component had to deal with implementing safe script execution into the web server. This means that we had to detect that the following request from the client was calling a PHP script. If it was, we would execute the script and take its output and send it to the client.

Some minor components include threading the web server, allowing it to handle multiple and simultaneous connections at time so that the implementation did not seem slow to the client. GNU TLS functions in this case for encryption can sometimes be fairly slow, this is why multi-threading was required.

2 Usage

Very simplistic usage, to run the application.

```
./web-server
```

Put all your files in the content directory, the web server will serve content only in that directory.

Place your browser to `https://localhost:8443/index.html`

3 Implementation

3.1 Web Server Implementation

For clarification purposes, this section will talk about how we handle the data and not the actual file descriptors themselves because due to the SSL implementation, GNU TLS handles most of the client file descriptors for us already. For example, a typical web call would call send() to the client, but in this case we call gnutls-record-send() in order to properly send over encrypted data to the pipe. The inputs are very similar except for the file descriptor. In a normal case the file descriptor would be an integer while in the SSL case, it is of a type gnutls-session-t.

When all the established connections are completed, we take in the request headers from the client browser using gnutls-record-recv (which is called by the function readline). We should note that read-line takes in one character at a time and appends it to a line buffer. Because of the time limitations, we only take in the first line which contains the file request header. This header looks something like ”GET /index.html HTTP/1.1”.

After readline has finished giving us our request header, we need to process it accordingly. We do this by splitting up the header by ”spaces”, using a string tokenizer. The first part of the header is the request type. Request types include GET, POST, PUT and DELETE. However, due to the nature of time, the only request type supported is GET. To implement POST and other parameters, we must have to process the request of header and properly organize it accordingly until we get to the data parameter for POST. So for now, we are limited to GET.

The second part of the header gives us which file to request. For example, this can be ”/index.html” or any complex path such as ”/other/index.html”. We also took into consideration that if ”/” was passed, we would take on index.html by default. We then pre-pended the content directory path into the request, making sure the web server wasn't allowed to grab anything else into the directory. Note by default ”..” in request parameters are disallowed in web browsers. Because of the strangeness of web browsers using connection keep-alive, we had to implement the response header of ”Content-Length:” in bytes. At the same time we detect the file size of the requested file, we also detect if the file exists or not. If it does not, we stop everything we're doing and just send a ”HTTP/1.1 404 Not found” header as described in the RFC 2616 and 4229 documentation for web servers. If the file does exist, we then find the file size with strlen() and append that to the request header of ”Content-Length: 638”, 638 being an example length.

We then need to tell the client browser how to analyze the content we need to give it. By RFC documentation, we need to send it the response header of ”Content-Type:”. To do so in a simplistic manner, we took the request file and processed the end extention and tried matching it with a list of known types. In our case, the known types we gave our server was HTML, Javascript, PNG, JPG and PHP. We then had to translate these file types into the mime type is a file identifier docuemnted in the RFC 2046 documentation and can be summarized at `http://en.wikipedia.org/wiki/Internet_media_type`. An example mime

type would be ”text/html” or ”application/javascript”. Once we’ve determined the content type, we append another response header such as ”Content-Type: application/javascript”.

By this point, we have completed the request header build for our web server. We then send this to the browser to interpret, following the actual content of the file along witha ”HTTP/1.1 200 OK” header. From there, we do not close the connection and let the browser handle the closing of the connection due to the default of ”keep-alive” with the browsers. Now it is known that we should properly handle this by interpreting the request header, but due to the time restrictions this was not possible.

3.2 SSL Encryption Implementation

In order to begin a SSL connection, we needed to generate self-signed certificates for the web server to give and serve to the browser. To do so, we followed instructions similar to this web page. `http://www.akadia.com/services/ssh_test_certificate.html`.

In the main function of our program, we first setup our TLS connection and its default parameters. Its like setting up the socket types in web browsers. But in this case, we have to define the type of certificates we’re using, the type of connection it will be using and more. Now the server requires a key and certificate. However, there are some intermediate certificates and so on that can be used (if there’s a third party involved in the verification). But in this case, we will not be using them.

After we set the TLS stage, we then create a normal web socket, using the default way of creating them. The only difference is when we receive requests from the client browser. When the browser sends us a request, we accept the request and make a file descriptor based on it. We then set the TLS pointer to the file descriptor where it will check if it requires an SSL connection. If the browser did not request a secure connection, it will fail and stop there. If it does, then we perform the proper handshake by sending the certificate to the browser for verification. When the verification is complete, a secure connection is established between the client and the server.

From here, we use our normal ways of passing and receiving data through the file descriptors, except we call GNU TLS functions instead, which will handle our decryption and encryption for us. This concludes the SSL encryption of the web server where it’ll terminate the GNU TLS connection between the browser and server.

3.3 PHP Execution

During the file type handling of the web server implementation, we detect if the script is PHP. If the script is of mime type ”application/php” we close the current file descriptor which opened the file, and create a different one in its place. In fact there is a nice function in C called popen which will execute a command line in ”/bin/sh” and throw it output into a file descriptor. From there on, we change the mime type into ”text/html” since the output should be of that type and we parse the file descriptor normally. Now if there’s a script error, the web server by default

will send the errors to the browser. To disable this you would need to edit PHP's configuration file.

4 Conclusion and Remarks

This entire application was built by myself, I hope you can understand the time constraint of the project and how I implemented many different parts of a full project that a team would build. I had a ton a fun building this web server and I hope you don't penalize me for the fact that I wanted to work by myself on this assignment.

5 Ammendments

Because of the limitations of GNUTLS, unfortunately we had to chunk the data if we were sending files larger than 16KB. To do so we chunk it up and send bit by bit.