CS1680 Final Project

Kenta Yoshii

December 7, 2022

Abstract

In my final project, I implemented a simple HTTP 1.1 webserver with very basic CGI, TLS, and mTLS supported. To view the source code, visit here

1 Introduction

ken-http is a simple webserver that I built to better understand how the HTTP protocol and all the other cool stuff, such as TLS, CGI, and mTLS, works. There are two parts to this final project.

1. In part 1, I implemented a simple webserver that supports the subset of HTTP/1.1. It supports GET, HEAD, and POST. To really understand how this really works, the implementation does not use the famous net/http package that Golang has. Rather, I made this possible by solely relying on sockets.
2. In part 2, I implemented various features that come with the HTTP protocol that I wanted to learn more in depth, which are: Common Gateway Interface, Transport Layer Security, and Mutual Transport Layer Security. Since I really wanted to understand these concepts better, I stuck with the basics. Since I already understood the workings of webserver and how they could be implemented with sockets, in this part, I used the net/http package so that some tasks could be omitted.

2 Design/Implementation

2.1 Part 1: HTTP server

2.1.1 Parsing HTTP Request Header

In order to understand the HTTP protocol in depth, instead of relying on existing package to parse the Request Header, I implemented my own decoder so that I could handle incoming HTTP requests on my server. I achieved this by parsing in a top-down manner, in which I went form: Request line (ex. GET / HTTP/1.1), Request Header(Header-name-1: value...), and Request Body (some content). However, I encountered some difficulty in parsing the Request Header since it was pretty hard to handle the different state machines. Hence, I decided to use a reference implementation.

2.1.2 Server

When starting up the server, you also pass in the path the resources that you want the server to serve alongside the server binary. To prevent repetitive marshalling from happening, I converted all the resources to bytes and store it in a custom struct so that whenever that resource was requested, I can just write that bytes to the client. Whenever a POST request was received, I will update the bytes with the body bytes.

2.1.3 Handling Requests

Whenever AcceptTCP returns, we launch a goroutine to handle each request. Then, in the handle client function, I parse the request line sent by the client and handle the request accordingly. Aftermarshalinge handling the requests, we send back a response to the client with the appropriate status code indicating the state of the request.

2.1.4 Common Gateway Interface

I also explored CGI in this project where we give an interface between a webserver and a script that generate a web content so that with CGI you could make the webserver dynamic. In order to achieve this, I used a package in Go, net/http/cgi. Then all I had to do was create a mapping between an endpoint and a script that will generate a webcontent once that endpoint was hit. Since this was a basic implementation, I made a simple bash script that returned a html content.

2.2 Part 2: TLS and mTLS

2.2.1 Certificate and key

For certificates, since I was not able to replicate certificates signed by trusted certificate authority, I made a public key that was self-signed. From there, I used the crypto package in Golang to generate a key pair with valid curve. Then I created a certificate template. Here the serial number is just a random integer. The template is copied from RFC 5280. Then signed template with the key generated earlier. Finally, I serialize the key and teh certificate into two .pem files so that we could use them later.

2.2.2 mTLS

I extend the above idea to provide client authentication, making this a mutual TLS. I generate clientkey and clientcert and load clientcert to the server to be added to the certPool.

3 Discussion/Results

3.1 Part 1: Webserver

To measure the performance of the simple webserver implementation using sockets, I used wrk which is a HTTP benchmarking tool capable of generating a bunch of traffic even when running on a single multi-core CPU. Below is a result from when I run a benchmark for 30 seconds, using 12 threads, and keeping 400 HTTP connections open. From the above image, we can see that our simple webserver

```
cs1680-user@ba952fe75b8b:~/final-project-kenta/wrk$ ./wrk -t12 -c400 -d30s http://localhost:8080
Running 30s test @ http://localhost:8080
  12 threads and 400 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   806.19ms   87.95ms   1.67s    92.32%
    Req/Sec    55.83     53.69   212.00     76.79%
  14577 requests in 30.10s, 10.02MB read
Requests/sec:    484.21
Transfer/sec:    340.94KB
```

Figure 1: wrt benchmarking

handled requests pretty well.

3.2 Part 2: TLS

For TLS, we first run the modified TLS-supported server. Then from another terminal, we connect to the server using curl without any options. Below is the result from that We get this because of the self-signed certificate we are using. By adding –cacert with path to the cert.pem, we obtain the below result which tells us that TLS is correctly implemented.

```
cs1680-user@ba952fe75b8b:~$ curl https://localhost:4000
curl: (60) SSL certificate problem: unable to get local issuer certificate
More details here: https://curl.haxx.se/docs/sslcerts.html

curl failed to verify the legitimacy of the server and therefore could not
establish a secure connection to it. To learn more about this situation and
how to fix it, please visit the web page mentioned above.
```

Figure 2: TLS error

```
cs1680-user@ba952fe75b8b:~/final-project-kenta$ curl -Lv --cacert ./cert.pem https://localhost:4000
*   Trying 127.0.0.1:4000...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 4000 (#0)
* ALPN, offering h2
* ALPN, offering http/1.1
* successfully set certificate verify locations:
*   CAfile: ./cert.pem
  CApath: /etc/ssl/certs
* TLSv1.3 (OUT), TLS handshake, Client hello (1):
* TLSv1.3 (IN), TLS handshake, Server hello (2):
* TLSv1.3 (IN), TLS handshake, Encrypted Extensions (8):
* TLSv1.3 (IN), TLS handshake, Certificate (11):
* TLSv1.3 (IN), TLS handshake, CERT verify (15):
* TLSv1.3 (IN), TLS handshake, Finished (20):
* TLSv1.3 (OUT), TLS change cipher, Change cipher spec (1):
* TLSv1.3 (OUT), TLS handshake, Finished (20):
* SSL connection using TLSv1.3 / TLS_AES_128_GCM_SHA256
```

Figure 3: success TLS error

3.3 Part 3: mTLS

Finally, for mTLS, we first check the need for mutual verification. Using curl with the same option as above, we see the following output on the server terminal This is good! This means the client needs

```
cs1680-user@ba952fe75b8b:~/final-project-kenta$ ./http-server-mtls
2022/12/06 18:45:01 Starting server on :4000
2022/12/06 18:45:14 http: TLS handshake error from 127.0.0.1:55042: tls: client didn't provide a certificate
```

Figure 4: mTLS error

to verify itself which forms the basis of mutual TLS. Passing the correct flag in, we see that it works now

```
cs1680-user@ba952fe75b8b:~/final-project-kenta$ curl -Lv --cacert ./cert.pem --cert ./clientcert.pem --key ./clientke
em https://localhost:4000
*   Trying 127.0.0.1:4000...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 4000 (#0)
* ALPN, offering h2
* ALPN, offering http/1.1
* successfully set certificate verify locations:
*   CAfile: ./cert.pem
  CApath: /etc/ssl/certs
* TLSv1.3 (OUT), TLS handshake, Client hello (1):
* TLSv1.3 (IN), TLS handshake, Server hello (2):
* TLSv1.3 (IN), TLS handshake, Encrypted Extensions (8):
* TLSv1.3 (IN), TLS handshake, Request CERT (13):
* TLSv1.3 (IN), TLS handshake, Certificate (11):
* TLSv1.3 (IN), TLS handshake, CERT verify (15):
* TLSv1.3 (IN), TLS handshake, Finished (20):
* TLSv1.3 (OUT), TLS change cipher, Change cipher spec (1):
* TLSv1.3 (OUT), TLS handshake, Certificate (11):
```

Figure 5: success mTLS

4 Future Work

The below are the Future work I can think of

1. Fully implement HTTP/1.1 or the latest HTTP version with all the Methods supported
2. Make the CGI more robust with more dynamic scripting.
3. Implement TLS with real certificates not self-signed one.
4. Combine everything I did into one big server

5 Conclusions

Through this project, I was not only able to understand the ins and outs of the HTTP protocol but also actively engage with the surrounding protocols such as Common Gateway Interface, Transport Layer Security, and Mutual Transport Layer Security.