# From HTTP to HTTPS with Go

2019-08-02 :: Prakhar Srivastav

#http #https #golang #tls #security

## Introduction⌗

In this post, we will learn how to configure TLS encryption in Go. We will further explore how to set mutual-TLS encryption. The code presented in this blog post is available here. In this post, we just show the relevant snippets. The interested readers can clone the repository and follow along.

We will start by writing a simple Http server and a client in Go. We will then encrypt the traffic between them by configuring TLS on the server. Towards the end of this post, we will configure mutual TLS between the two parties.

## A Simple http server⌗

Let's start by creating an Http client-server implementation in Go. We expose an Http endpoint `/server` reachable on `localhost:8080` . Then we call the endpoint using `http.Client` and print the result.

Full implementation is availabe here.

```
1   // Server code
2   mux := http.NewServeMux()
3   mux.HandleFunc("/server", func(w http.ResponseWriter, r *http.Req
4     fmt.Fprint(w, "Protect Me ... ")
5   })
6   log.Fatal(http.ListenAndServe(":8080", mux))
7
8
9   // Client code
```

```
10    if r, err = http.NewRequest(http.MethodGet, "http://localhost:808
11      log.Fatalf("request failed : %v", err)
12    }
13
14    c := http.Client{
15      Timeout:   time.Second * 5,
16      Transport: &http.Transport{IdleConnTimeout: 10 * time.Second},
17    }
18
19    if data, err = callServer(c, r); err != nil {
20      log.Fatal(err)
21    }
22    log.Println(data) // Should print "Protect Me ... "
```

In the next sections, we will encrypt the traffic between the client and the server using TLS. Before we come to that stage, we should set up our public key infrastructure (PKI).

## PKI Setup⌗

To set up our mini PKI infrastructure, we will use a Go utility called minica to produce root, server, and the client keypairs and certificates. In reality, a Certificate Authority (CA) or a Domain Administrator (within an organization) will provide you a keypair and a signed certificate. In our case, we will use minica to provision this for us.

### Generating keypair and certificates⌗

**Note**: If generating these seems a hassle, you can reuse the certificates committed with the Github repository.

We will use the below steps to generate certificates.

1. Install minica: `go get github.com/jsha/minica`
2. Create server certificate by running `minica --domains server-cert`
3. If you are running it for the first time, it will generate 4 files.

   a. minica.pem (root certificate)
   b. minica-key.pem (private key for root)
   c. server-cert/cert.pem (certificate for domain “server-cert”, signed by root’s public key)
   d. server-cert/key.pem (private key for domain “server-cert”)
4. Create client certificate by running `minica --domains client-cert .` It will generate 2 new files

   a. client-cert/cert.pem (certificate for domain “client-cert”)
   b. client-cert/key.pem (private key for domain “client-cert”)

Alternatively, you can also use IP instead of domains with minica to generate your keypairs and certificates.

### Setting alias in /etc/hosts⌗

The client and the server certificates generated above are valid for the domains `server-cert` and `client-cert` respectively. These domains do not exist, so we will create an alias for `localhost` (127.0.0.1). Once this is set up, we will be able to access our Http server using `server-cert` instead of `localhost` .

If you are on a platform other than Linux, you should Google on how to set it up for your OS. I use a Linux machine and setting the domain alias is pretty straightforward. Open `/etc/hosts` file and add below entries.

```
1    127.0.0.1       server-cert
2    127.0.0.1       client-cert
```

At this point, our infrastructure setup is complete. In the next sections, we will configure the server with these certificates, to encrypt the traffic between the client and the server.

## Configuring TLS on the server⌗

Let use the key and certificate generated for the `server-cert` domain to configure TLS on the server. The client is the same as earlier. The only difference is that we will call the server on three different URLs to understand what is going on under the hood.

Full implementation is here

```
// Server configuration
mux := http.NewServeMux()
mux.HandleFunc("/server", func(w http.ResponseWriter, r *http.Req
    fmt.Fprint(w, "i am protected")
})
log.Println("starting server")
// Here we use ListenAndServerTLS() instead of ListenAndServe()
// CertPath and KeyPath are location for certificate and key for
log.Fatal(http.ListenAndServeTLS(":8080", CertPath, KeyPath, mux)

// Server configuration
c := http.Client{
    Timeout:   5 * time.Second,
    Transport: &http.Transport{IdleConnTimeout: 10 * time.Second,
}

if r, err = http.NewRequest(http.MethodGet, "http://localhost:808
//if r, err = http.NewRequest(http.MethodGet, "https://localhost:
//if r, err = http.NewRequest(http.MethodGet, "https://server-cer
    log.Fatalf("request failed : %v", err)
}

if data, err = callServer(c, r); err != nil {
    log.Fatal(err)
}
log.Println(data)
```

We start the server using `http.ListenAndServeTLS()` that takes four arguments, port, the path to the public certificate, the path to private key and Http-handler. Let us examine the response from the

server. We send three different requests that will fail but will give us more insight into how Http encryption works.

1. Attepmt 1 to `http://localhost:8080/server` , the response is:

   **Client Error**: *Get http://localhost:8080/server: net/http: HTTP/1.x transport connection broken: malformed HTTP response "\x15\x03\x01\x00\x02\x02"*

   **Server Error**: *http: TLS handshake error from 127.0.0.1:35694: tls: first record does not look like a TLS handshake*

   This is good news, which means the server is sending encrypted data. No one over Http will be able to make sense of it.

2. Attempt 2 to `https://localhost:8080/server` , the response is:

   **Client Error**: *Get https://localhost:8080/server: x509: certificate is valid for server-cert, not localhost*

   **Server Error**: *http: TLS handshake error from 127.0.0.1:35698: remote error: tls: bad certificate*

   This is again good news, this means that a certificate issued to domain `server-cert` cannot be used by other domains (localhost).

3. Attempt 3 to `https://server-cert:8080/server` , the response is:

   **Client Error**: *Get https://server-cert:8080/server: x509: certificate signed by unknown authority*

   **Server Error**: *http: TLS handshake error from 127.0.0.1:35700: remote error: tls: bad certificate*

   This error demonstrates that the client does not trust signed that certificate. Clients must be aware of the CA which has signed the certificate.

The whole idea behind this section was to demonstrate three guarantees that TLS ensures:

- The message is always encrypted.
- The server is actually what it says it is.
- The client should not blindly believe the server certificate. They should be able to verify the server's identity through a CA.

## Configuring CA certificates on the client⌗

Let us configure the CA certificates on the client so that it can verify the server's identity against root CA's certificate. Since server-cert's certificate was signed using root CA's public key, the TLS handshake will validate and the communication will be encrypted.

Full implementation is available here.

```
// create a Certificate pool to hold one or more CA certificates
rootCAPool := x509.NewCertPool()

// read minica certificate (which is CA in our case) and add to t
rootCA, err := ioutil.ReadFile(RootCertificatePath)
if err != nil {
    log.Fatalf("reading cert failed : %v", err)
}
rootCAPool.AppendCertsFromPEM(rootCA)
log.Println("RootCA loaded")

// in the http client configuration, add TLS configuration and ad
c := http.Client{
    Timeout: 5 * time.Second,
    Transport: &http.Transport{
        IdleConnTimeout: 10 * time.Second,
        TLSClientConfig: &tls.Config{RootCAs: rootCAPool},
    },
}

if r, err = http.NewRequest(http.MethodGet, "https://server-cert:
    log.Fatalf("request failed : %v", err)
}

if data, err = callServer(c, r); err != nil {
    log.Fatal(err)
}
log.Println(data)

// server response
prakhar@tardis (master)X % go run client.go
RootCA loaded
i am protected # response from server
```

This ensures all the three guarantees that we discussed earlier.

## Configuring mutual TLS⌗

We have established a client's trust on the server. But in a lot of use cases, the server needs to trust the client. For example, financial, healthcare or public service industry. For these scenarios, we can configure mutual TLS between the client and the server so that both parties can trust each other.

The TLS protocol has support for this from the beginning. The steps required to configure mutual TLS authentication are as follows:

1. The server gets its certificate from a CA (CA-1). The client should have a public certificate of CA-1 that has signed the server's certificate.
2. The client gets its certificate from a CA (CA-2). The server should have the public certificate of CA-2 that has signed the client's certificate. For simplicity, we will use the same CA (CA-1 == CA-2) to sign both client and server certificates.
3. The server creates a CA certificate pool to validate all the clients. At this point, the server includes a public certificate of CA-2.
4. Similarly, the client creates its own CA certificate pool and includes a public certificate for CA-1.
5. Both parties validate the incoming requests against the CA certificate pool. If there are any validation errors on either side, the connection will be aborted.

Let us see it in action. Full implementation for this functionality is available here

### Server configuration⌗

```
1    mux := http.NewServeMux()
```

```
2   mux.HandleFunc("/server", func(w http.ResponseWriter, r *http.Req
3       fmt.Fprint(w, "i am protected")
4   })
5
6   clientCA, err := ioutil.ReadFile(RootCertificatePath)
7   if err != nil {
8       log.Fatalf("reading cert failed : %v", err)
9   }
10  clientCAPool := x509.NewCertPool()
11  clientCAPool.AppendCertsFromPEM(clientCA)
12  log.Println("ClientCA loaded")
13
14  s := &http.Server{
15      Handler: mux,
16      Addr:    ":8080",
17      TLSConfig: &tls.Config{
18          ClientCAs:  clientCAPool,
19          ClientAuth: tls.RequireAndVerifyClientCert,
20          GetCertificate: func(info *tls.ClientHelloInfo) (certific
21              c, err := tls.LoadX509KeyPair(CertPath, KeyPath)
22              if err != nil {
23                  fmt.Printf("Error loading key pair: %v\n", err)
24                  return nil, err
25              }
26              return &c, nil
27          },
28      },
29  }
30  log.Fatal(s.ListenAndServeTLS("", ""))
```

There are a few key things to note in this configuration:

1. Instead of using `http.ListenAndServeTLS()`, we use `server.ListenAndServerTLS()`.
2. We load the server certificate and key inside `tls.Config.GetCertificate` function.
3. We create a pool of client CA certificates that the server should trust.
4. We configure `tls.Config.ClientAuth = tls.RequireAndVerifyClientCert`, which will always verify the

certificate of all the clients that try to connect. Only the validated clients will be able to continue the conversation.

## Client settings⌗

The `http.Client` configuration changes a little for the client as well.

```
rootCA, err := ioutil.ReadFile(RootCertificatePath)
if err != nil {
    log.Fatalf("reading cert failed : %v", err)
}
rootCAPool := x509.NewCertPool()
rootCAPool.AppendCertsFromPEM(rootCA)
log.Println("RootCA loaded")

c := http.Client{
    Timeout: 5 * time.Second,
    Transport: &http.Transport{
        IdleConnTimeout: 10 * time.Second,
        TLSClientConfig: &tls.Config{
            RootCAs: rootCAPool,
            GetClientCertificate: func(info *tls.CertificateReque
                c, err := tls.LoadX509KeyPair(ClientCertPath, Cli
                if err != nil {
                    fmt.Printf("Error loading key pair: %v\n", er
                    return nil, err
                }
                return &c, nil
            },
        },
    },
}
```

Notice some of the differences in the configuration as compared to server:

1. In tls.Config, we use `RootCAs` to load certificate pool against `ClientCAs` setting on the server.
2. We use `tls.Config.GetClientCertificate` to load client certificates against `tls.Config.GetCertificate` on the server.

The actual code in GitHub provides some callbacks, which could be used to see certificate information as well.

## Running mutual TLS authenticated client and server⌗

```
# Server logs
2019/08/01 20:00:50 starting server
2019/08/01 20:00:50 ClientCA loaded
2019/08/01 20:01:01 client requested certificate
Verified certificate chain from peer:
  Cert 0:
    Subject [client-cert] # Server shows the client certificate d
    Usage [1 2]
    Issued by minica root ca 5b4bc5
    Issued by
  Cert 1:
    Self-signed certificate minica root ca 5b4bc5

# Client logs
2019/08/01 20:01:01 RootCA loaded
Verified certificate chain from peer:
  Cert 0:
    Subject [server-cert] # Client knows the server certificate d
    Usage [1 2]
    Issued by minica root ca 5b4bc5
    Issued by
  Cert 1:
    Self-signed certificate minica root ca 5b4bc5
2019/08/01 20:01:01 request from server
2019/08/01 20:01:01 i am protected
```

## Conslusion⌗

TLS configuration has always been more of a certificate management problem rather than an implementation affair. The typical confusions in the TLS configuration are often around using the correct certificates rather than its implementation. If you understand the TLS protocol and handshake correctly, Go offers everything else you need right out of the box.

You should also check an earlier post where we explored the TLS encryption and security from a theoretical standpoint.

## References⌗

This post is hugely inspired by this wonderful talk by Liz Rice in Gophercon-2018, please check it out. Other useful references are mentioned below:

- secure-connections: repo for gophercon talk by Liz Rice.
- minica Certificate Authority.
- this amazing article by Eric Chiang. A must-read.
- step-by-step-guide-to-mtls-in-go.
- this article on medium.

---

READ OTHER POSTS

← GCP - Using KMS to ma… Remoto RPC Framework →

Github | LinkedIn