Dynamically update TLS certificates in a Golang server without downtime

Configuring an HTTPS server for automatically updated certificates is not as hard as you might think.

By Savita Ashture

October 6, 2022 | 0 Comments | 5 min read

Register or Login to like

Image by: *Opensource.com*

Transport Layer Security (TLS) is a cryptographic protocol based on SSLv3 designed to encrypt and decrypt traffic between two sites. In other words, TLS ensures that you're visiting the site you meant to visit and prevents anyone between you and the website from seeing the data being passed back and forth. This is achieved through the mutual exchange of digital certificates: a private one that exists on the web server, and a public one typically distributed with web browsers.

In production environments, all servers run securely, but server certificates may expire after some period. It is then the server's responsibility to validate, regenerate, and reuse newly generated certificates without any downtime. In this article, I demonstrate how TLS certificates are updated dynamically using an HTTPS server in Go.

These are the prerequisites for following this tutorial:

- A basic understanding of client-server working models
- A basic knowledge of Golang

Configuring an HTTP server

Before I discuss how to update certs dynamically on an HTTPS server, I'll provide a simple HTTP server example. In this case, I need only the `http.ListenAndServe` function to start an HTTP server and `http.HandleFunc` to register a response handler for a particular endpoint.

To start, configure the HTTP server:

```
package main

import(
        "net/http"
        "fmt"
        "log"
)

func main() {

        mux := http.NewServeMux()
        mux.HandleFunc("/", func( res http.ResponseWriter, req *http.Request ) {
                fmt.Fprint( res, "Running HTTP Server!!" )
        })

        srv := &http.Server{
                Addr: fmt.Sprintf(":%d", 8080),
                Handler:           mux,
        }

        // run server on port "8080"
        log.Fatal(srv.ListenAndServe())
}
```

More for sysadmins

- Enable Sysadmin blog
- The Automated Enterprise: A guide to managing IT with automation
- eBook: Ansible automation for Sysadmins

In the example above, when I run the command `go run server.go`, it will start an HTTP server on port 8080. By visiting the http://localhost:8080 URL in your browser, you will be able to see a Hello World! message on the screen.

The `srv.ListenAndServe()` call uses Go's standard HTTP server configuration. However, you can customize a server using a `Server` structure type.

To start an HTTPS server, call the `srv.ListenAndServeTLS(certFile, keyFile)` method with some configuration, just like the `srv.ListenAndServe()` method.
The`ListenAndServe` and `ListenAndServeTLS` methods are available on both the HTTP package and the `Server` structure.

The `ListenAndServeTLS` method is just like the `ListenAndServe` method, except it will start an HTTPS server.

```
func ListenAndServeTLS(certFile string, keyFile string) error
```

As you can see from the method signature above, the only difference between this method and the `ListenAndServe` method is the additional `certFile` and `keyFile` arguments. These are the paths to the *SSL certificate file* and *private key* file, respectively.

Generating a private key and an SSL certificate

Follow these steps to generate a root key and certificate:

1. Create the root key:

```
openssl genrsa -des3 -out rootCA.key 4096
```

2. Create and self-sign the root certificate:

```
openssl req -x509 -new -nodes -key rootCA.key -sha256 -days 1024 -out rootCA.crt
```

Next, follow these steps to generate a certificate (for each server):

1. Create the certificate key:

```
openssl genrsa -out localhost.key 2048
```

2. Create the certificate-signing request (CSR). The CSR is where you specify the details for the certificate you want to generate. The owner of the root key will process this request to generate the certificate. When creating the CSR, it is important to specify the `Common Name` providing the IP address or domain name for the service, otherwise the certificate cannot be verified.

```
openssl req -new -key localhost.key -out localhost.csr
```

3. Generate the certificate using the TSL CSR and key along with the CA Root key:

```
 openssl x509 -req -in localhost.csr -CA rootCA.crt -CAkey rootCA.key -CAcreateser
```

Finally, follow the same steps for generating certificates for each server to generate certificates for clients.

Configuring an HTTPS server

Now that you have both private key and certificate files, you can modify your earlier Go program and use the `ListenAndServeTLS` method instead.

```
package main

import(
        "net/http"
        "fmt"
        "log"
)

func main() {

        mux := http.NewServeMux()
        mux.HandleFunc("/", func( res http.ResponseWriter, req *http.Request ) {
                fmt.Fprint( res, "Running HTTPS Server!!" )
        })

        srv := &http.Server{
                Addr: fmt.Sprintf(":%d", 8443),
                Handler:           mux,
        }

        // run server on port "8443"
        log.Fatal(srv.ListenAndServeTLS("localhost.crt", "localhost.key"))
}
```

When you run the above program, it will start an HTTPS server using `localhost.crt` as the `certFile` and `locahost.key` as the `keyFile` from the current directory containing the program file. By visiting the https://localhost:8443 URL via browser or command-line interface (CLI), you can see the result below:

```
$ curl https://localhost:8443/ --cacert rootCA.crt --key client.key --cert client.c

Running HTTPS Server!!
```

That's it! This is what most people have to do to launch an HTTPS server. Go manages the default behavior and functionality of the TLS communication.

Configuring an HTTPS server for automatically update certificates

While running the HTTPS server above, you passed `certFile` and `keyFile` to the `ListenAndServeTLS` function. However, if there is any change in `certFile` and `keyFile` because of certificate expiry, the server needs to be restarted to take those effects. To overcome that, you can use the `TLSConfig` of the `net/http` package.

The `TLSConfig` structure provided by the `crypto/tls` package configures the TLS parameters of the `Server`, including server certificates, among others. All fields of the `TLSConfig` structure are optional. Hence, assigning an empty struct value to the `TLSConfig` field is no different than assigning a `nil` value. However, the `GetCertificate` field can be very useful.

```
type Config struct {
   GetCertificate func(*ClientHelloInfo) (*Certificate, error)
}
```

The `GetCertificate` field of the `TLSConfig` structure returns a certificate based on the given `ClientHelloInfo` .

I need to implement `GetCertificate` closure function which uses `tls.LoadX509KeyPair(certFile string, keyFile string)` or `tls.X509KeyPair(certFile []byte, keyFile []byte)` function to get the certificates.

Now I'll create a `Server` struct that uses `TLSConfig` field value. Here is a sample format of the code:

```
package main

import (
        "crypto/tls"
        "fmt"
        "log"
        "net/http"
)

func main() {

        mux := http.NewServeMux()
        mux.HandleFunc("/", func( res http.ResponseWriter, req *http.Request ) {
                fmt.Fprint( res, "Running HTTPS Server!!\n" )
        })

        srv := &http.Server{
                Addr: fmt.Sprintf(":%d", 8443),
                Handler:           mux,
                TLSConfig: &tls.Config{
                        GetCertificate: func(*tls.ClientHelloInfo) (*tls.Certificat
                                                // Always get latest localhost.crt a
                                                // ex: keeping certificates file som
                                cert, err := tls.LoadX509KeyPair("localhost.crt", "
                                if err != nil {
                                        return nil, err
                                }
                                return &cert, nil
                        },
                },
        }

        // run server on port "8443"
        log.Fatal(srv.ListenAndServeTLS("", ""))
}
```

In the above program, I implemented the `GetCertificate` closure function, which returns a cert object of type `Certificate` by using the `LoadX509KeyPair` function with the certificate and private files created earlier. It also returns an error, so handle it carefully.

Since I am preconfiguring server instance `srv` with a TLS configuration, I do not need to provide `certFile` and `keyFile` argument values to the `srv.ListenAndServeTLS` function call. When I run this server, it will work just like before. But this time, I have abstracted all the server configuration from the invoker.

```
$ curl https://localhost:8443/ --cacert rootCA.crt --key client.key --cert client.c

Running HTTPS Server!!
```

Final thoughts

A few additional tips by way of closing: To read the cert and key file inside the `GetCertificate` function, the HTTPS server should run as a goroutine. There is also a StackOverflow thread that provides other ways of configuration