Golang
Creating and Using Certificates with TLS

CHRIS MCENIRY

Chris "Mac" McEniry is a practicing sysadmin responsible for running a large e-commerce and gaming service. He's been working and developing in an operational capacity for 15 years. In his free time, he builds tools and thinks about efficiency. cmceniry@mit.edu

In this article, we're going to extend Kelsey's original work from Spring 2016 *;login:* on the gls service [1]. To recap, gls is a distributed ls tool, which calls out to a listening service to perform a directory listing. One of the open items left from that article is the concern around authentication and authorization. To extend that, we're going to add secured authentication to both sides of the gls tool and with this we're going to gain a minimal amount of authorization.

The ubiquitous Internet connection security protocol is currently Transport Layer Security (TLS). TLS is used to encrypt, authenticate, and authorize (to a degree) connections. The defaults handle encryption for us well enough, so in this article, we're going to examine authentication and authorization. Authentication is based on the names on exchanged certificates that have been signed by third party certificate authorities. Once identity has been established, the service can then incorporate a base level of authorization based on the names (e.g., parsing user=$username so it will get access to items specific to $username) on the certificates or on the certificate chain (e.g., this was signed by the "users" CA, so it will get access to common user items).

In our example, we want to ensure four items: encrypted communication, successful identification of the glsd server (that the one gls connects to is the proper one), successful identification of the gls client (that the one that connects to the glsd server is the proper one), and restricted access of the gls client as appropriate. To accomplish this, we're going to add TLS between the client and the server, enable verification on both server and client, and compare the certificate identity to a good list. In order to support all of this, we need to first generate some private keys and certificates for gls and glsd to use.

NOTE: We've cut some corners to simplify the example in this article. Several additional areas should be considered in a full production PKI infrastructure, including, but not limited to, use of intermediate CAs, revocation lists, full subjects, selection of hash, key properties, private key encryption with a passphrase, etc.

Certificates

In terms of authentication, TLS is a form of public key cryptography. If you're not familiar with it, you can read Radia Perlman's *;login:* article about Bitcoin [2]. The issue with plain public key cryptography is that you have to distribute the public keys. Instead of having to distribute every certificate for every service to every potential user of that service, TLS builds a chain of trust in the same way that a Web browser authenticates a Web site like a bank or hospital.

When I use a browser to connect to a Web site, the site sends my browser a certificate. This certificate has the Web site's public key and a subject name that identifies the Web site, and it is signed by a trusted third party called the certificate authority (CA). My browser has a bundle of certificate authorities, and it looks for a match for the signature in that bundle. If there isn't a match, the browser will alert about an untrusted certificate. With a matching

signature, the browser can verify that the Web site's certificate has been issued by the CA, and so the browser trusts it. In this way, the browser doesn't have to have the certificate for the Web site ahead of time but only needs to have a much smaller set of certificate authorities to use to verify.

After the chain of trust has been used to verify that the Web site's certificate is valid, the browser does another check. This time, it takes the subject name on the certificate and compares that to the DNS name that the browser used to connect. If the certificate name does not match the DNS name, the browser will alert to a name mismatch. If it does match, the browser trusts the Web site and proceeds.

This chain of trust can be used to authenticate the client side as well, with one caveat. The Web server can require that my browser supplies a certificate as well, and it can compare the signature on that certificate to its bundle of trusted certificate authorities. In most cases, this is for an internal or private situation, so there's only one certificate authority to check against, but uses can vary. However, a DNS check of the client is unlikely to work in many cases: multiple clients behind a Network Address Translation, residential networks, or networks behind dynamic addressing are all unlikely to be able to issue certificates appropriately to match the actual end client. Therefore, the server is very unlikely to check the name on the certificate in the same way as the client does to authenticate the server. The server uses the certificate in two ways: the name on the certificate can be used to identify the user or provide a group or role; and the fact that the certificate is signed is often used to provide a base level of authorization ("if it's signed, it's allowed in").

Since this is a private service, we can consider that our certificate authority handling and chain handling is working together. That allows us to only produce three certificates: a common certificate authority, a server certificate, and a client certificate. The server certificate will get the localhost name since that is what is being used to connect to; and we're going to encode a username, glss Client A into the client certificate to show a stronger authentication approach than just verifying the certificate.

Building on the gls Package with the glss Package

Before we start, we need a place to work that isn't conflicting with previous work. We want to use the existing work of the RPC mechanisms in the gls package and only add the pieces that we need. We're going to use the built-in package manager `go get` to pull in Hightower's work, and augment this with our own working path. For article space, the full code is not in this article, but it is available on GitHub [3]. You can pull in the final source code for this exercise along with the original source code. If you want to assemble the code yourself, this article steps through that, but you will have to fill in some of the gaps. To get started down that path:

```
$ go get github.com/kelseyhightower/gls
$ mkdir -p $GOPATH/src/github.com/cmceniry/login-glss
$ cd $GOPATH/src/github.com/cmceniry/login-glss
$ mkdir -p certs server client
```

Otherwise, you can pull in the new code along with the original:

```
$ go get github.com/kelseyhightower/gls
$ go get github.com/cmceniry/login-glss
```

`go get` will place the gls package at $GOPATH/src/github.com/kelseyhightower/gls. We will be referencing it in our import statements much as we do for the standard library utilities:

```
import (
   "fmt"
   "github.com/kelseyhightower/gls"
)
```

Instead of using the utilities gls and glsd in the existing gls package, we're going to create three new utilities in the login-glss package: `client/main.go` and `server/main.go`, to hold the service like before but with TLS encryption, and a new command, `certs/main.go`, which we'll next use to generate our keys and certificates.

Generating Keys and Certificates

As a private service, we're going to handle all of the certificate and certificate authority management internally. In a production case, this may work, or you may want to use a commercial vendor or Let's Encrypt [4]—the process for obtaining certificates and keys is slightly different, but we'll end up with the same resulting items. In addition, since this is again internal, we're going to use one certificate authority for the client and the server certificate signing. Since this exercise is on Go, we're going to generate these using Go itself. Let's start this by opening a new file:

```
certs/generate_certs.go
```

The Go standard crypto library has all of the functions needed to generate certificate/key pairs. We'll want to import these libraries and some other ones that we'll be using into our file:

```
package main

import (
  "crypto/rand"
  "crypto/rsa"
  "crypto/x509"
  "crypto/x509/pkix"
  "encoding/pem"
  "io/ioutil"
  "math/big"
  "time"
)
```

Since we have three keys and certificates to generate, we're going to wrap this process up into a single function, `generateKeyAndCert`. This function takes in a subject name and the certificate and key of a certificate authority. We can use the same function for our certificate authority, and in that case, nil can be passed for the `signer` and `signerkey`.

```
func generateKeyAndCert(
  name string,
  signer *x509.Certificate,
  signerkey *rsa.PrivateKey,
) (
  *rsa.PrivateKey,
  *x509.Certificate,
) {
```

`generateKeyAndCert`'s body has four parts to it. First, we have to generate the private key/public key pair. As mentioned, a key is a set of cryptographic numbers, in this case represented as an `rsa.PrivateKey` [5] struct. The inputs to it are limited—a random number source, which we're using as the default, and a key length. Later, we'll be using one of the fields of the `key`, the paired `PublicKey`, to generate the certificate.

```
key, _ := rsa.GenerateKey(rand.Reader, 2048)
```

Second, we must generate a template x509.Certificate [6]. It might be a bit confusing, but the template is of type `x509.Certificate`, which is the same type that we'll receive at the end. The template is used by the standard library function to generate certificates for where to source all of the information that we'll need. There are a few required fields: SerialNumber (unique distinguisher), Subject (which is where we're going to push CommonName), NotBefore/NotAfter (which determine the lifetime of this certificate), and KeyUsage (the intended purpose of this certificate).

```
template := &x509.Certificate{
    SerialNumber: big.NewInt(1),
    Subject:      pkix.Name{CommonName: name},
    NotBefore:    time.Now().Truncate(24 * time.Hour),
    NotAfter:     time.Now().Truncate(24 * time.Hour).
      Add(365 * 24 * time.Hour),
    KeyUsage:     x509.KeyUsageKeyEncipherment |
      x509.KeyUsageDigitalSignature,
  }
```

Since this is a dual purpose function, we might be generating a certificate authority. In those cases, we need to set a couple of additional fields: `IsCA` must be true, and `KeyUsage` must be extended for this additional purpose. Additionally, we also need to set our currently nil-valued `signer` and `signerkeys`. As a root CA, we're going to set these to themselves.

```
if signer == nil || signerkey == nil {
  template.IsCA = true
  template.KeyUsage |= x509.KeyUsageCertSign
  signer = template
  signerkey = key
}
```

Next, we're ready to generate our certificate using the standard library function: `x509.CreateCertificate`. In addition to the default source for random numbers, it uses the template, the signer, our newly generated public key, and the signer's private key to create a binary blob representing the signed certificate.

```
der, _ := x509.CreateCertificate(
  rand.Reader,
  template,
  signer,
  &key.PublicKey,
  signerkey,
)
```

And, finally, we need to make this binary blob useful. This binary blob is DER encoded [7]. While this is useful to functions handling binary data, we want to force the structure and type consistency of the language and turn this into a full certificate datatype.

```
cert, _ := x509.ParseCertificate(der)
```

We now have the actual key and cert, so we can pass those back:

```
  return key, cert
}
```

Once we generate these, we'll need to be able to save them to disk to be used by our client and server utilities. The standard format for handling key and certificate files is called privacy-enhanced electronic mail (PEM; https://en.wikipedia.org/wiki/Privacy-enhanced_Electronic_Mail) encoding. The PEM is an ASCII form generated from the binary data, held as an array of bytes in Go, of the keys and certificates. Extracting the binary data is slightly different for keys and certificates, but both need to be converted over to this PEM format, and there are standard library functions available for this. Once we get the PEM form in memory, we can dump this to disk using the convenient `ioutil.WriteFile` function.

```
func saveKeyAndCert(
  prefix string,
  key *rsa.PrivateKey,
  cert *x509.Certificate,
) {
  keyBytes := x509.MarshalPKCS1PrivateKey(key)
  keyPem := pem.EncodeToMemory(
    &pem.Block{Type: "RSA PRIVATE KEY", Bytes: keyBytes})
```

```
  ioutil.WriteFile(prefix+".key", keyPem, 0444)
  certPem := pem.EncodeToMemory(
    &pem.Block{Type: "CERTIFICATE", Bytes: cert.Raw})
  ioutil.WriteFile(prefix+".crt", certPem, 0444)
}
```

With our wrapping and save-to-disk functions, we can put together our main function. Note the use of the CA keys and certificates to generate the actual end keys and certificates:

```
func main() {
  caKey, caCert := generateKeyAndCert(
    "glss Root CA",
    nil, nil)
  saveKeyAndCert(
    "certs/CA", caKey, caCert)
  serverKey, serverCert := generateKeyAndCert(
    "localhost",
    caCert, caKey)
  saveKeyAndCert(
    "certs/server", serverKey, serverCert)
  clientKey, clientCert := generateKeyAndCert(
    "glss Client A",
    caCert, caKey)
  saveKeyAndCert(
    "certs/client", clientKey, clientCert)
}
```

With this utility written, we're now ready to execute it. Since this is a one-time tool for this exercise, let's just run it:

```
$ go run certs/generate_certs.go
```

You should see several certificate and key files in the certs directory:

```
CA.crt
CA.key
client.crt
client.key
server.crt
server.key
```

Now that we have all of the certificates, we can proceed into encryption and authenticating our communications.

Server Changes

Part of what makes this powerful in Go is that we won't have to change much code to wrap the calls in TLS. We can change some pieces of the setup to include TLS setup, and the rest of the application is unchanged. Part of this is because we're able to swap out different types that satisfy the same Go interface—in particular `net.Conn` on the server side.

Start by copying the original server and client utilities from the gls package.

```
$ cp \
  $GOPATH/src/github.com/kelseyhightower/gls/server/main
  go \
  ./server/main.go
```

We're going to start by updating the import list. We have to add specific crypto libraries that we're going to be using as well as add back in the reference to the original gls library.

```
import (
  ...
  "crypto/tls"
  "crypto/x509"
  "io/ioutil"

  "github.com/kelseyhightower/gls"
)
```

Next, we need to initialize the TLS settings for the server. This involves three parts: loading the server key pair, loading the certificate authority certificate to verify against, and then using those to set the TLS configuration. To load the key pair, we will use the `tls.LoadX509KeyPair` function.

```
func main() {
  cert, err := tls.LoadX509KeyPair("certs/server.crt",
    "certs/server.key")
  if err != nil {
    log.Println(err)
    return
  }
```

TLS connections are verified against a `CertPool`, which is a list of certificate authorities used to check for signatures. In the case of verifying against a wide range of certificate authorities, like a browser would do, you can keep adding certificate authorities to the pool. In this case, we only have our internal certificate, so we can add only it to the `CertPool`. Since the certificate authority is a bare certificate (i.e., it doesn't include a private key), we can't use `tls.LoadX509KeyPair` to get the certificate; we have to load it separately and then add it bare to the `CertPool`.

```
caCert, err := ioutil.ReadFile("certs/CA.crt")
if err != nil {
  log.Fatal(err)
}
caCertPool := x509.NewCertPool()
caCertPool.AppendCertsFromPEM(caCert)
```

Now with the server certificate and the certificate authority, we can set the TLS configuration. In addition to the certificates, we want to require that we authenticate the client using TLS.

```
config := &tls.Config{
  Certificates: []tls.Certificate{cer},
  ClientCAs:    caCertPool,
  ClientAuth:   tls.RequireAndVerifyClientCert,
}
```

As we'll see in the client, Go has a convenience function inside of TLS for connections; for the server, `tls.Listen` can replace `net.Listen`. However, we need to be able to access the peer information, so we have to set up TLS directly and can't use this. Luckily, this only requires a couple of lines (plus error checking): one to create the TLS connection object, and one to perform the TLS handshake.

```
for {
  conn, err := l.Accept()
  if err != nil {
    log.Println(err)
  }
  tlsconn := tls.Server(conn, config)
  err = tlsconn.Handshake()
  if err != nil {
    log.Fatal(err)
  }
```

Once the TLS handshake is successful, we can inspect the connection for the client information and confirm it is correct. Note that we may get multiple certificates on the connection. A client may send its full certificate chain or a partial certificate chain over the connection if it needs to connect intermediate certificates to a root. The key here is that first certificate (index 0) will be the leaf certificate for *this* client, so it will be the one we check against. In our particular case, we're going to compare the subject's `CommonName`, but other situations could use other fields of the certificate.

```
  tlsclient := tlsconn.ConnectionState().PeerCertificates[0]
  if tlsclient.Subject.CommonName != "glss Client A" {
    log.Fatal("Invalid client")
  }
  log.Printf("user=\"%s\" connect",
    tlsclient.Subject.CommonName)
```

Now that we've verified the certificate chain (via the `ClientAuth` setting on `tls.Config`) and checked that the `CommonName` is correct, we can proceed with the `net/rpc` call. Special Note: since this is providing a wrapper layer, we're going to insert this between the `Accepted` connection and `rpc.ServConn`. `Accept` and `tls.Server` both return `net.Conn`, and `rpc.ServConn` takes in a `net.Conn`. `rpc.ServConn` isn't aware that the data is being encrypted underneath it.

```
  rpc.ServConn(tlsconn)
  conn.Close()
}
```

You can confirm everything by building the server the same as before:

```
$ go build -o glssd server/main.go
```

At this point, we've added TLS to the server side without having to change any of the underlying `net/rpc` items. Now we need to do the same on the client side.

Client Changes

The client changes are the same as on the server side except that we don't have to check anything additional on the certificate's `CommonName`—this is handled by default when TLS authenticates servers. As before, start by copying the existing gls client over to our new working directory:

```
$ cp \
  $GOPATH/src/github.com/kelseyhightower/gls/client/main.
go \
  ./client/main.go
```

Then update the imports the same as before.

```
import (
  ...
  "crypto/tls"
  "crypto/x509"
  "io/ioutil"

  "github.com/kelseyhightower/gls"
)
```

Next, load the client certificate and private key, the certificate authority certificate, and configure TLS. The main differences are to flip from authentication of the clients to authentication of the server in the `tls.Config`: we're not specifying `ClientAuth`, since that's a server side optional setting, and we're specifying the `RootCAs` instead of `ClientCAs` to indicate that we're connecting out and authenticating the server instead of being connected to and authenticating the client.

```
cert, err := tls.LoadX509KeyPair("certs/client.crt",
  "certs/client.key")
if err != nil {
  log.Fatal(err)
}
caCert, err := ioutil.ReadFile("certs/CA.crt")
if err != nil {
  log.Fatal(err)
}
caCertPool := x509.NewCertPool()
caCertPool.AppendCertsFromPEM(caCert)
conf := &tls.Config{
  Certificates: []tls.Certificate{cert},
  RootCAs:      caCertPool,
}
```

Next, we connect to the server with the convenience function `tls.Dial`, and pass the returned `net.Conn` to `rpc.NewClient`. In the same way as encryption and authentication are transparent on the server, this is transparent to `net/rpc` on the client.

```
conn, err := tls.Dial("tcp", "localhost:8080", conf)
if err != nil {
  log.Fatal(err)
}
client := rpc.NewClient(conn)
```

Build the client, and you should now have a fully encrypted and authenticated gls client:

```
$ go build -o glss client/main.go
```

Start up the server and, separately in another terminal, start up the client:

```
$ ./glssd
# In another terminal
$ ./glss ~
```

Conclusion

At the end of this, we have protected the gls connection with mutual TLS authentication. In addition, we've relied on the power of the golang interface to only make minimal changes to the original program to enable secure communication.

References

[1] K. Hightower, "Modern System Administration with Go and Remote Procedure Calls (RPC)," *;login:*, vol. 41, no. 1 (Spring 2016), pp. 63–67: https://www.usenix.org/publications/login/spring2016/hightower

[2] R. Perlman, "Blockchain: Hype or Hope?" *;login:*, vol. 42, no. 2 (Summer 2017): https://www.usenix.org/publications/login/summer2017/perlman.

[3] Source code for glss: https://github.com/cmceniry/login-glss.

[4] Let's Encrypt: https://letsencrypt.org.

[5] Go Doc on PrivateKey: https://godoc.org/crypto/rsa#PrivateKey.

[6] Go Doc on x509: https://godoc.org/crypto/x509#Certificate.

[7] DER encoding: https://en.wikipedia.org/wiki/X.690#DER_encoding.