Go (https://youngkin.github.io/tags/go) Distributed Applications (https://youngkin.github.io/tags/distributed-applications) HTTPS (https://youngkin.github.io/tags/https) TLS (https://youngkin.github.io/tags/tls)

Create Secure Clients and Servers in Golang Using HTTPS

Posted on Sunday, September 20, 2020

Contents

Overview

The purpose of this article is to show how to write secure web services and clients using Go and HTTPS. In researching how to accomplish this I came across numerous articles and gists. However, none of them provided the complete picture I needed to implement a robust client or server. Most of them only provided terse code examples with equally terse examples regarding how to create the certificates needed for the code to work. Others were oriented to gRPC or plain TLS over TCP. I also wanted to understand what I was doing, not just the syntax. The point of this article is to provide not only the *how*, but also the *why’s* behind the *how*.

If you’re new to HTTPS, TLS, and public/private keys you might want to read the following *aside*.

ASIDE:

Security is paramount to ensure the privacy and well being of customers. They want to be certain that the information they provide, like passwords and credit card numbers, are going to the expected service provider. Likewise, it is sometimes necessary for service providers to ensure that they are communicating with the expected customer.

There are two characteristics of secure communications:

- *Trust*
- *Encryption*

Trust is the foundation of security. Without trust there can be no assurance that the parties to a conversation are not bad actors. Encryption is required to ensure that bad actors can't listen in on a conversation and gain access to sensitive information or perform harmful actions. The following requirements must be met to ensure trust and security:

- *Trusted authorities must exist that can vouch for the identify of a client or a server*
- *Trusted sources of encryption technology that can be used by clients and servers to encrypt their communications must also exist*
- *Support in the code for the associated techniques and technologies that are used to implement client and server applications*

The umbrella for all these things is called the Public Key Infrastructure(PKI) (https://en.wikipedia.org/wiki/Public_key_infrastructure). There are four components to PKI that implement the requirements outlined above:

- ***Public Key Encryption** - A public key can be safely shared by the owner, with another party, that provides the other party with the ability to encrypt and decrypt a conversation. The owner of a public key has a corresponding private key that is not shared. A public key can only decrypt information that was encrypted with the corresponding private key. Likewise, the private key is required to decrypt information that was encrypted with a public key. This asymmetric characteristic of the encryption guarantees that information can be securely shared between known parties to a conversation.*
- ***Certificates** - contain the identity of the holder as well as the certificate owner's public key. A certificate also includes information about the trusted authority that issued the certificate.*
- ***Certificate Authorities(CA)** - are the trusted source of identity information. They can also issue certificates or delegate that authority to a Registration Authority.*
- ***Registration Authority (RA)** - are a trusted source of certificates. Any certificates issued by an RA will also contain the certificate of the CA that authorized them to issue certificates.*

This is required to ensure the trust requirement can be met. In other words, RAs themselves are only trusted because they can prove that they're trusted by a CA.

Create certificates and keys

The basis for proving identity and encrypting information is a certificate and a corresponding Certificate Authority (CA). For the purposes of this article we'll need both. There are two ways to go about obtaining these, the hard way and the easy way. The hard way is appropriate for real world applications. It involves registering a domain (e.g., `youngkin.com`), obtaining DNS services for that domain, and obtaining a certificate for that domain. While not difficult it will require a fair amount of work and you'll probably have to spend some money to register the domain.

An easier way to provide a realistic experience is to create your own CA and obtain a certificate from this CA. This article uses CA signed certificates vs. self-signed certificates in order to create that more realistic experience. I found creating a CA, requesting certificates, and having the CA sign those certificates helpful in understanding the entire process. Usually servers access CA certificates installed on the machine. This article will demonstrate how to register a CA certificate programmatically.

There are at least a couple of tools available help us with the easy way, certstrap (https://github.com/square/certstrap) and easy-rsa (https://github.com/OpenVPN/easy-rsa).

The definitive command line tool for working with certificates is called `openssl`. It can be used to create CAs, RAs, and certificates, as well as do many other useful things (see the references section for more details). It can also be used to create what are called self-signed certificates. Creating a Self-Signed SSL Certificate (https://linuxize.com/post/creating-a-self-signed-ssl-certificate/) is a good resource for how to do this. While it is oriented towards Linux, there are versions of `openssl` available on Windows and Mac machines. I recommend installing the Libre fork of `openssl`. The reasoning behind this is well described in a StackExchange question/answer (https://security.stackexchange.com/questions/112545/what-are-the-main-advantages-of-using-libressl-in-favor-of-openssl). To verify whether you have Libre `openssl` run the following:

```
$ openssl version
LibreSSL 2.8.3
```

Output like the above indicates Libre `openssl` is being used.

Several files will be created during the certificate generation process, some with `.crt` and `.key` suffices. You may also see the suffix `.pem` when reading about certificates. It's worth noting that `.pem` files are equivalent to `.crt` and `.key` files. **PEM** is a file format. `.crt` and `.key` are hints as to what the file contains (certificates and keys), but these files all use the PEM format. See this StackOverflow discussion (https://stackoverflow.com/questions/62823792/how-to-get-crt-and-key-from-cert-pem-and-key-pem) for more details about this.

Install software to create the CA and certificate(s)

As mentioned above there are at least 2 options available to easily create a CA and a CA's registered certificates. This article will use `certstrap` for no other reason than it was written in Go. However, `easy-rsa` is a good alternative.

Before starting, download the appropriate executable from the certstrap releases (https://github.com/square/certstrap/releases) page on GitHub. I placed mine in my `~/bin` directory which is in my PATH. You'll also need to make it executable (`chmod +x <downloadedfilename>`).

We'll follow the usage instructions (https://github.com/square/certstrap) from the project's README. At the end of this section we will have created a CA, a certificate and key for our server, and a certificate and key for our client. I keep all my certificates in a directory called `~/certs` . All the following commands will be run from that directory. When prompted for the `passphrase` just hit enter (i.e., no passphrase).

Create the CA

```
~/certs certstrap init --common-name "ExampleCA"
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Created out/ExampleCA.key
Created out/ExampleCA.crt
Created out/ExampleCA.crl
```

`init` directs `certstrap` to create a new CA. `--common-name` (CN) specifies the name for the our CA, which is named *ExampleCA*. Three files are created:

- `ExampleCA.key` is the private key for *ExampleCA*
- `ExampleCA.crt` is the certificate for *ExampleCA*

- `ExampleCA.crl` is the certificate revocation list (CRL) for that CA. It contains a list of revoked certificates issued by the associated CA.

> *A **Common Name** or CN is typically the fully qualified domain name (FQDN) of the host associated with a certificate (not strictly true for client certificates (https://jamielinux.com/docs/openssl-certificate-authority/sign-server-and-client-certificates.html)). See this source (https://jamielinux.com/docs/openssl-certificate-authority/certificate-revocation-lists.html) for information on certificate revocation lists.*

Create the certificates for the client and servers

```
~/certs certstrap request-cert --domain  "localhost"
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Created out/localhost.key
Created out/localhost.csr

~/certs certstrap request-cert --domain  "client"
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Created out/client.key
Created out/client.csr
```

> *`localhost` is used as the domain for the server since, as noted above, a valid FQDN of the host is required for servers. `localhost` suffices for this purpose. Using the `--domain` flag will create a Subject Alternative Name (SAN) in addition to the CN in the certificate signing request (csr) and in the certificate itself. SANs are the current standard for specifying a number of methods for addressing a service, including by FQDN or domain name.*
> *There's an alternative to `--domain`, `--common-name`. We used `--common-name` to create the CA above. `--common-name` will only generate a CN. Starting in Go 1.15 certificates must contain a SAN entry or the https request will fail. Certificates with only a CN will not be accepted. If Go 1.15 or higher is used, and `--common-name` is used to generate the CSR, you will likely see the following error from the client:*
>
> ```
> Get "https://localhost": x509: certificate relies on legacy Common Name field, use SANs or t
> ```
>
> *As noted in the error message, this problem can be overcome by prefixing the client command with `GODEBUG=x509ignoreCN=0`.*

Sign the certificates for the client and servers

```
~/certs certstrap sign localhost --CA ExampleCA
Created out/localhost.crt from out/localhost.csr signed by out/ExampleCA.key

~/certs certstrap sign client --CA ExampleCA
Created out/client.crt from out/client.csr signed by out/ExampleCA.key
```

> *Please note that all certificates and associated keys were placed in the `./out` directory.*

Certificates must be signed by a trusted authority, in this case the CA, in order to be valid. Signing is a guarantee by the CA that the owner of the certificate is who they say they are. The `--CA` flag above directs `certstrap` to have the certificates signed by our Exa,mpleCA.

At this point we have certificates and keys for the CA, the client, and the server.

Write the Client and Server

All code in this article is available at GitHub in my gohttps (https://github.com/youngkin/gohttps) repository.

We'll write both a simple server that does no validation against a client's certificate, as well as a more advanced server that is capable of a variety of options when validating a client's certificate. Finally, we'll develop a client that can talk to both servers.

Before moving on we need to briefly discuss how HTTPS is implemented. HTTPS traffic is encrypted by the TLS layer. TLS is the successor to SSL and works on top of TCP/IP. It does a number of things including:

- Negotiates the TLS session. This involves negotiating the version of TLS and the encryption suite to be used.
- Validates the server's identity
- If required, it validates the client's identity
- Handles all traffic encryption

Go’s HTTP package includes a TLS configuration struct that is used to implement a client’s and server’s HTTPS communication expectations. This will be a focus in the following sub-sections.

> *TLS requires a reliable transport mechanism. TCP and UDP are the 2 choices, but UDP is not reliable. See Why does TLS require TCP (https://security.stackexchange.com/questions/170833/why-does-tls-require-tcp) for more discussion on this topic.*

A simple server

See GitHub (https://github.com/youngkin/gohttps/blob/master/simpleserver/server.go) for the complete implementation of the simple server.

The simplest HTTPS interaction between a client and a server is one where the client validates the server’s credentials and where all traffic is encrypted. The client only requires access to the certificate for the CA that signed the server’s certificate. The server neither knows nor cares about the client’s identity. This is a pretty common use case.

Here’s a breakdown of the implementation of a very simple HTTPS server. The first thing to do is create and configure the `http.Server` struct:

```
server := &http.Server{
    Addr:         ":" + *port,
    ReadTimeout:  5 * time.Minute, // 5 min to allow for delays when 'curl' on OSx prompts for use
    WriteTimeout: 10 * time.Second,
    TLSConfig:    &tls.Config{ServerName: *host},
}
```

`Addr` simply specifies the listening address for the server. `ReadTimeout` and `WriteTimeout` set the timeouts for reads and writes respectively. As the comment indicates, `ReadTimeout` is set to 5 minutes to allow time for the entry of the machine’s user’s password. OSx sometimes prompts for this when `curl` is used and the certificate is password protected. In this article none of the certificates are password protected.

> *Strictly speaking the timeout fields aren't needed in a simple server such as this. That said, robust servers will include them. See The complete guide to Go net/http timeouts (https://blog.cloudflare.com/the-complete-guide-to-golang-net-http-timeouts/) for more details.*

In the context of this article, `TLSConfig` is the most interesting field in the `Server` struct. This is where all the TLS options are configured. In this case only `ServerName` is required. `ServerName` must match the hostname in the server's certificate.

The server needs a handler function:

```
http.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
    body, err := ioutil.ReadAll(r.Body)
    if err != nil {
        body = []byte(fmt.Sprintf("error reading request body: %s", err))
    }
    resp := fmt.Sprintf("Hello, %s from Simple Server!", body)
    w.Write([]byte(resp))
})
```

There isn't much to this handler function, it provides just the common "Hello, World!" response.

The final step is for the server to begin listening for requests:

```
if err := server.ListenAndServeTLS(*serverCert, *srvKey); err != nil {
    log.Fatal(err)
}
```

Instead of the `ListenAndServe` call in an HTTP server, an HTTPS server uses `ListenAndServeTLS`. `*serverCert` and `*srvKey` are the server's certificate and private key files respectively. The filenames containing these are passed in on the command line (more on that below). As you may recall, in this article `localhost.crt` and `localhost.key` are the certificate and key files we created for the servers.

A more advanced server

See GitHub (https://github.com/youngkin/gohttps/blob/master/advserver/server.go) for the complete implementation of this more advanced server.

The primary difference between the simple server above and a more secure server is the addition of the capability to require, or, require and validate, a client's certificate.

As before, an `http.Server` struct is required:

```
server := &http.Server{
    Addr:         ":" + *port,
    ReadTimeout:  5 * time.Minute, // 5 min to allow for delays when 'curl' on OSx prompts for use
    WriteTimeout: 10 * time.Second,
    TLSConfig:    getTLSConfig(*host, *caCert, tls.ClientAuthType(*certOpt)),
}
```

Notice in this version the `TLSConfig` field is being configured by a function, `getTLSConfig()`. There's a bit more to this `TLSConfig` than in the simple server case. Here's the code for the function:

```
func getTLSConfig(host, caCertFile string, certOpt tls.ClientAuthType) *tls.Config {
        var caCert []byte
        var err error
        var caCertPool *x509.CertPool
        if certOpt > tls.RequestClientCert {
                caCert, err = ioutil.ReadFile(caCertFile)
                if err != nil {
                        log.Fatal("Error opening cert file", caCertFile, ", error ", err)
                }
                caCertPool = x509.NewCertPool()
                caCertPool.AppendCertsFromPEM(caCert)
        }

        return &tls.Config{
                ServerName: host,
                ClientAuth: certOpt,
                ClientCAs:  caCertPool,
                MinVersion: tls.VersionTLS12, // TLS versions below 1.2 are considered insecure -
        }
}
```

Let’s break this down by sections. First let’s take a look at the function signature.

```
func getTLSConfig(host, caCertFile string, certOpt tls.ClientAuthType) *tls.Config
```

The arguments are as follows:

- `host` - this is the server’s hostname. It must match the name provided in the host’s certificate. In our case this is the SAN.
- `caCertFile` - this is the certificate file name for the CA that signed the ***client’s*** certificate, in this case `ExampleCA.crt` . The CA’s certificate is required in this server because we created an unknown CA, i.e., not a CA that’s normally configured in the OS (e.g., the KeyChain in OSx). So we need to add it here.
- `certopt` - as can be seen this is of type `tls.ClientAuthType` There are 5 authorization types for authorizing/validating a client’s certificate:
    - `tls.NoClientCert` - A client certificate will not be requested and it is not required. This is the default value.
    - `tls.RequestClientCert` - A client certificate will be requested, but it is not required and it won’t be validated

- `tls.RequireAnyClientCert` - A client certificate is required, but any valid client certificate is acceptable. It will not be validated against the CA's certificate.
- `tls.VerifyClientCertIfGiven` - A client certificate will not be requested, but if present it will be validated against the CA's certificate
- `tls.RequireAndVerifyClientCert` - A client certificate will be required and will be validated against the CA's certificate

Now that we know what the function's arguments are let's take a look at the function body:

```
1   var caCert []byte
2   var err error
3   var caCertPool *x509.CertPool
4   if certOpt > tls.RequestClientCert {
5       caCert, err = ioutil.ReadFile(caCertFile)
6       if err != nil {
7           log.Fatal("Error opening cert file", caCertFile, ", error ", err)
8       }
9       caCertPool = x509.NewCertPool()
10      caCertPool.AppendCertsFromPEM(caCert)
11  }
12
13  return &tls.Config{
14      ServerName: host,
15      ClientAuth: certOpt,
16      ClientCAs:  caCertPool,
17      MinVersion: tls.VersionTLS12,
```

- Line 3 - we define an `x509.CertPool`. This is a pool of certificates that will be used below. It will contain the certificate of the CA that signed the client's certificate.
- Line 4 - we check the value of the `certOp`. Any value above `tls.RequestClientCert` will require clients to provide a certificate.
- Lines 5 - 8 - In order to validate client certificates a CA certificate needs to be loaded into the `caCertPool`. These lines read the CA certificate file and handle any errors
- Lines 9 -10 - We create a new `x509.CertPool` and add the CA's certificate to the pool.
- Line 15 - The `ClientAuth` field is used to specify the level of client certificate authorization and validation that's required. The values and definitions were given above in the discussion about `certOpt`.

- Line 16 - The `ClientCAs` field is used to specify the CAs that will be used to validate client certificates. It's value is set from the `caCertPool` that was created on lines 9 & 10. The GoDoc for this field contains the following:
  - *ClientCAs defines the set of root certificate authorities that servers use if required to verify a client certificate by the policy in ClientAuth.*
- Line 17 - `MinVersion` sets the minimum TLS version to accept when negotiating versions with the client. TLS versions below 1.2 are considered insecure. See RFC 7525 (https://www.rfc-editor.org/rfc/rfc7525.txt) for details.

Create the client

See GitHub (https://github.com/youngkin/gohttps/blob/master/client/client.go) for the complete implementation of this client.

As stated earlier, this client can successfully communicate with either the simple or advanced servers. Let's take a look at the significant code.

The first thing to do is to configure the client's certificate and key if present.

```
if *clientCertFile != "" && *clientKeyFile != "" {
    cert, err = tls.LoadX509KeyPair(*clientCertFile, *clientKeyFile)
    if err != nil {
        log.Fatalf("Error creating x509 keypair from client cert file %s and client key file %s",
    }
}
```

If provided, the code will create an `x509` keypair from the client's certificate and private key. This keypair will be used when negotiating the TLS connection and for encrypting and decrypting communications between the client and server.

Next, as with the advanced server, we'll create the certificate pool that will contain the certificate of the CA that signed, in this case, the server's certificate.

```
caCert, err := ioutil.ReadFile(*caCertFile)
if err != nil {
    log.Fatalf("Error opening cert file %s, Error: %s", *caCertFile, err)
}
caCertPool := x509.NewCertPool()
caCertPool.AppendCertsFromPEM(caCert)
```

Then we create an `http.Transport` with the client's certificate/keypair and the certificate pool containing the CA. This `http.Transport` will be used to configure the `http.Client`.

```
t := &http.Transport{
    TLSClientConfig: &tls.Config{
        Certificates: []tls.Certificate{cert},
        RootCAs:      caCertPool,
    },
}

client := http.Client{Transport: t, Timeout: 15 * time.Second}
```

`http.Transport` contains a `tls.Config`. As with the servers, the `Certificates` field is populated with the client's certificate. There is a new field here, the `RootCAs` field. The GoDoc describes this field as follows:

> *RootCAs defines the set of root certificate authorities that clients use when verifying server certificates. If RootCAs is nil, TLS uses the host's root CA set.*

This is everything that needs to be done to prepare the client to interact with an HTTPS server. The remaining code prepares and sends the request and processes the response.

Putting it all together

If you've cloned or forked the gohttps (https://github.com/youngkin/gohttps) repository you'll notice that it has the following directory structure:

```
gohttps
      |
      +- advserver
      +- client
      +- simpleserver
```

The source files for the client and each of the servers are in their respective directories. The programs can be built and run from these directories.

The certificates and keys referenced in the command lines below match the names of the ones generated in the "Create Certificates and Keys" section above.

Start server

Simple server command line (no client validation)

```
./simpleserver -host "localhost" -srvcert "/path/to/localhost.crt" -srvkey "/path/to/localhost.key
```

Advanced server command line will full client certificate validation (`-certopt 4`).

```
./advserver -host "localhost" -srvcert "/path/to/localhost.crt" -cacert "/path/to/ExampleCA.crt" \
-srvkey "/path/to/localhost.key" -port 443 -certopt 4
```

Run the client

Using `curl`

`curl` request with no client certificate validation.

```
curl -vi -d "World" --cacert /path/to/ExampleCA.crt https://localhost
```

`curl` request with full client certificate validation.

```
curl -d "World" -vi --cert /path/to/client.crt  --key /path/to/client.key --cacert ./out/ExampleCA
```

Using the client program

Command line for full client certificate validation by the server.

```
./client -clientcert "/path/to/client.crt" -clientkey "/path/to/client.key" -cacert "/path/to/Exam
```

Command line for no client certificate validation by the server.

```
./client -cacert "/path/to/ExampleCA.crt"
```

> *As noted in the section on creating certificates above, if the client and server certificates aren't created with the `--domain` flag you may see the following error:*
>
> ```
> Get "https://localhost": x509: certificate relies on legacy Common Name field, use SANs or t
> ```
>
> *Prefixing the previous commands with `GODEBUG=x509ignoreCN=0` will resolve the error. See the section on creating certificates for more details on how to avoid this problem.*

Conclusion

Thanks for following along. We covered a lot in this article including:

- If you had no background in PKI and certificates hopefully you learned enough to get you started on a journey to learn more about the subject
- A brief description of the different software tools available to work with certificates and keys. For this article we used `certstrap`.
- How to use `certstrap` to create a CA as well as client and server certificate signing requests, certificates, and keys.
- How to write a simple and a more advanced HTTPS server in Go, covering the specifics in detail
- How to write an HTTPS client in Go, again covering the specifics in detail

- Running the server programs and accessing the running processes using both `curl` and the client program

Feel free to use the code in the gohttps repository (https://github.com/youngkin/gohttps) for your own learning, experimentation, or as the basis to create your own clients and servers.

References

- Cover Photo by Yogesh Pedamkar (https://unsplash.com/@yogesh_7?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText) on Unsplash (https://unsplash.com/s/photos/lock?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText)
- The TLS Connection Options (https://github.com/jcbsmpsn/golang-https-example) GitHub project is a good resource to quickly create working HTTPS clients and servers. It was my primary source for information about this topic when I was first getting started in creating HTTPS clients and servers. It also covers solutions to some common problems.
- Secure gRPC with TLS/SSL (https://bbengfort.github.io/programmer/2017/03/03/secure-grpc.html) - although geared towards gRPC, the basic TLS underpinnings are the same as with HTTPS. This article filled in some gaps about how to configure TLS in Go.
- The Complete Guide To Switching From HTTP To HTTPS (https://www.smashingmagazine.com/2017/06/guide-switching-http-https/) provides a detailed, practical, discussion about almost all aspects of the technology behind HTTPS, how to request certificates, and how to configure various web servers to support HTTPS. It’s worth at least a quick perusal to see if there’s anything of interest.
- Public Key Infrastructure(PKI) (https://en.wikipedia.org/wiki/Public_key_infrastructure) is a mix of technology and trusted organizations that provide the underpinnings of secure communication
- TLS (https://en.wikipedia.org/wiki/Transport_Layer_Security) is an encryption protocol used to secure communications over the Internet
- HTTPS (https://en.wikipedia.org/wiki/HTTPS) is a secure implementation of the HTTP protocol
- OpenSSL Certificate Authority (https://jamielinux.com/docs/openssl-certificate-authority/index.html) is a good resource about creating CAs and certificates using `openssl`.
- Creating a Self-Signed SSL Certificate (https://linuxize.com/post/creating-a-self-signed-ssl-certificate/) is a good resource about creating self-signed certificates. These weren’t used in this article but it’s good to know that it’s possible to do this.

- Certstrap GitHub repo (https://github.com/square/certstrap)
- easy-rsa GitHub repo (https://github.com/OpenVPN/easy-rsa)
- QualSys SSL Server Test (https://www.ssllabs.com/ssltest/index.html) is a web page you can use to test HTTPS access to an HTTPS server and receive an evaluation of how well it implements best practices.
- Idrix (https://prod.idrix.eu/secure/) has a nice service to test SSL certificates. For example:
    - curl -vi –cert ./ClientCert.crt –key ./ClientCert.key https://prod.idrix.eu/secure/ (https://prod.idrix.eu/secure/)
- 21 OpenSSL Examples to Help You in Real-World (https://geekflare.com/openssl-commands-certificates/) is a good reference containing commonly used `openssl` commands

← PREVIOUS POST (HTTPS://YOUNGKIN.GITHUB.IO/POST/CREATEAFREEBLOGSITE/)

NEXT POST → (HTTPS://YOUNGKIN.GITHUB.IO/POST/SMOKEDBRISKET/)

13 Comments 11 ONLINE Sort By Best

Write your comment... LOGIN SIGNUP

A **Anil** 8 days ago
How install certstrap-1.2.0-linux-amd64 on linux?
Reply Share 1 0

Rich Youngkin 7 days ago
I followed the instructions at
https://github.com/square/certstrap
Reply Share 0 0

B **Bojan** 6 months ago
Excellent post with all steps in detail and useful links, thank you
Reply Share 1 0

G **Guest** 5 days ago
Thanks for the great tutorial, it is very useful and easy to digest.

One small thing: instead of downloading, building and then copying certstrap to the go bin directory manually, the exact same thing can be achieved by simply issuing:

```
go install github.com/square/certstrap@latest
```

Reply Share 0 0

Rich Youngkin 5 days ago
Excellent point, thanks for the suggestion!
Reply Share 0 0

M **Moree** 3 months ago
You're more than amazing, you made it as clear as day. I've benefited a greatly from this, and also kept it as a main reference that I would recommend to others.

Subscribe to stay up to date with my newsletter! (https://tinyletter.com/elev5280)

● () ● (mailto:rich.youngkin@gmail.com)

● (https://github.com/youngkin)

● (https://www.linkedin.com/in/richard-youngkin-0749763)

● (https://medium.com/@RichYoungkin)

● (https://stackoverflow.com/users/2646870/rich)

● (https://www.reddit.com/user/elevation5280)