

# CH18 加密安全

# 18-1 前言

- 在前面幾章，我們已經談過應用程式會面臨的一些安全性問題：
  - 資料庫的SQL注入攻擊(CH13)
  - 網頁的跨網站指令攻擊(CH15)
  - Goroutine的記憶體資源競爭(CH16)
- 本章將要討論另一個重要的安全議題：資料交換與網路通訊的加密

- 當你瀏覽網站時，可能會注意到有些網址前面是https://而非http://, 這是因為那些網站使用了TLS(Transport Layer Security, 傳輸層安全協定), 許多網站在處理敏感資料時，也會使用有簽署的數位憑證(digital certificate)來驗證身分和加密資訊
- Go語言有非常完整的標準加密套件，包含雜湊加密法，對稱/不對稱加密演算法，數位簽章和憑證等等
- 不過由於這個題材的範圍廣泛，本章只會挑選最適合的來示範

## 18-2 雜湊函式

- 雜湊(hashing)是把明碼(plaintext)文字用演算法轉成雜湊值(hash value), 其輸出結果理論上是獨一無二的
- 雜湊函式常用來檢查檔案的完整性：傳送者會根據檔案內容產生checksum(檢查碼/核對和), 接收者只要再產生一次checksum並和傳送者提供的雜湊值做比較, 就知道檔案是否正確
- 注意：雖然雜湊函式可用來加密資料, 但它和後面介紹的加密法其實是兩回事：雜湊是單向不可逆的過程, 加密法則可以用金鑰還原資料

- 儘管雜湊衝突(hash collision, 兩個不同值的雜湊值剛好相同)的機率很低, 但駭客仍能發起碰撞攻擊(collision attack), 也就是找到非法的值來使其檢查通過; 目前世界上常用的MD5和SHA1雜湊法就很容易遭到破解, 幸好新的雜湊演算法大大提高了安全性
- go語言的雜湊函式介面都大同小異, 以MD5為例, 我們可以使用內建套件crypto/md5的Sum()函式來替來源資料產生checksum:

Sum(in []byte) [<Size>]byte

Sum()傳回的是固定長度陣列, 其長度由MD5套件決定

- `crypto`模組也實作了SHA(安全雜湊演算法)家族的SHA1和SHA2 : SHA2(包括SHA256和SHA512)原理和SHA1類似，但安全性強很多
- 若想更進一步練習的話，你可以透過外部套件取得更強大的SHA3雜湊法

# 練習：使用不同的雜湊套件

- 下面我們會寫一個函式`getHash()`，並根據使用者傳入的雜湊函式名稱來呼叫對應的功能
- 但由於不同雜湊函式傳回的`byte`陣列長度不同，因此我們會先用`fmt.Sprintf()`把陣列轉成16進位數值，再以字串形式傳回
- 注意這個練習會用到外部套件<http://golang.org/x/crypto/sha3>；請參照第8章的`go get`下載並建立`go.mod`

```
1  package main
2
3  ∨ import (
4      "crypto/md5"
5      "crypto/sha256"
6      "crypto/sha512"
7      "fmt"
8
9      "golang.org/x/crypto/sha3" //外部套件 SHA3
10 )
11
12 ∨ func getHash(input, hashType string) string {
13 ∨     switch hashType {
14 ∨     case "MD5":
15         return fmt.Sprintf("%x", md5.Sum([]byte(input)))
16 ∨     case "SHA256":
17         return fmt.Sprintf("%x", sha256.Sum256([]byte(input)))
18 ∨     case "SHA512":
19         return fmt.Sprintf("%x", sha512.Sum512([]byte(input)))
20 ∨     case "SHA3_512":
21         return fmt.Sprintf("%x", sha3.Sum512([]byte(input)))
22 ∨     default:
23         return fmt.Sprintf("%x", sha512.Sum512([]byte(input)))
24     }
25 }
26
```



```
27  ✓ func main() {  
28      fmt.Println("[MD5]      :", getHash("Hello World!", "MD5"))  
29      fmt.Println("[SHA256]   :", getHash("Hello World!", "SHA256"))  
30      fmt.Println("[SHA512]   :", getHash("Hello World!", "SHA512"))  
31      fmt.Println("[SHA3_512]:", getHash("Hello World!", "SHA3_512"))  
32  }  
33
```

# 執行結果：

```
PS D:\git\GO\ch18\18-2> go run .  
[MD5]      : ed076287532e86365e841e92bfc50d8c  
[SHA256]   : 7f83b1657ff1fc53b92dc18148a1d65dfc2d4b1fa3d677284add200126d9069  
[SHA512]   : 861844d6704e8573fec34d967e20bcfef3d424cf48be04e6dc08f2bd58c729743371015ead891cc3cf1c9d34b49264b510751b1ff9e537937bc46b5d6  
ff4ecc8  
[SHA3_512]: 32400b5e89822de254e8d5d94252c52bdc27a3562ca593e980364d9848b8041b98eabe16c1a6797484941d2376864a1b0e248b0f7af8b1555a778c33  
6a5bf48
```

# 密碼管理：使用bcrypt雜湊函式

- 除了檢查檔案正確性，許多網站或資料庫也會將使用者的密碼轉成雜湊函式來儲存，這樣即使密碼的雜湊值外洩，外人也無法拿它推回密碼；而網站只要拿使用者輸入的字串雜湊值和密碼雜湊值比對，就能驗證密碼是否正確
- 不僅如此，go語言的外部套件 [golang.org/x/crypto/bcrypt](https://golang.org/x/crypto/bcrypt) 提供專門用來管理密碼的雜湊函式 `bcrypt`，且讓你用更簡單的方式檢查密碼：

```
1 package main
2
3 import (
4     "fmt"
5     "os"
6
7     "golang.org/x/crypto/bcrypt"
8 )
9
10 func main() {
11     password := "mysecretpassword"
12     fmt.Println("密碼明碼 :", password)
13
14     //用bcrypt將密碼轉成雜湊值
15     hash, err := bcrypt.GenerateFromPassword([]byte(password), bcrypt.DefaultCost)
16     if err != nil {
17         fmt.Println(err)
18         os.Exit(1)
19     }
20     fmt.Println("密碼雜湊值:", string(hash))
21
22     //測試輸入的新密碼是否符合
23     testString := "mysecretpassword"
24     err = bcrypt.CompareHashAndPassword([]byte(hash), []byte(testString))
25     if err != nil {
26         fmt.Println("密碼不符")
27     } else {
28         fmt.Println("密碼相符")
29     }
30 }
31
```

# 執行結果：

```
PS D:\git\GO\ch18\18-2.2> go run .
```

```
密碼明碼    : mysecretpassword
```

```
密碼雜湊值: $2a$10$KD6lVRhQN0VfC30YhZexXOKqIk0bG4m2BCHIyQUB78bWUdrurJb3m
```

```
密碼相符
```

## 18-3 加密法

- 真正的加密(**encryption**), 是將敏感資料加以轉換, 好讓收件人以外的對象無法閱讀, 只有收件人可用金鑰解密(**decryption**)
- 說不定你之前聽過靜態加密(**encryption at rest**)和傳輸中加密(**encryption in transit**): 前者是指資料在儲存前先行加密(比如存入資料庫), 後者則是在資料在傳輸過程中加密(比如在網路上傳送), 例如稍後會談到的**HTTP/TLS**

- 安群加密法本質上都很複雜，但很多都會對外公開，讓所有人都能使用；go語言提供了對稱式(**symmetric**)及非對者式(**asymmetric**)加密套件，本小節和下一節會來看看這兩種使用範例

## 18-3-1 對稱式加密

- 顧名思義，對稱加密法使用相同的金鑰來加密解密，而這金鑰會由通訊雙方共同持有
- Go語言提供兩種常見的對稱加密法：DES(資料加密標準)和AES(進階加密標準)
  - DES歷史較為悠久，也較容易破解，因此漸漸被AES取代



- 註：

- 可想而知，共用金鑰一外洩就會成為安全漏洞，不過對稱式加密法速度仍比較快，它在現今仍然很常用；那麼該如何安全交換金鑰呢？本章稍後介紹的**HTTP/TLS**就使用非對稱加密來傳送金鑰；另一個方式是使用迪菲-赫爾曼密鑰交換(**Diffie-Hellman key exchange**)協定，但這裡不多討論

- **DES和AES都屬於區塊加密法(block cipher)**, 意即它得將資料分隔成多個區塊後分別加密
- 針對各區塊的加密方式又有多種模式可用：以下練習我們就使用**AES**搭配安全性高的**GCM(Galois/Counter Mode, 伽羅瓦計數器模式)**
- 在**go**語言中, 使用**AES**加密和解密的步驟如下：

1. 用crypto/aes套件的NewCipher()函式產生一個區塊加密物件(cipher.Block結構)；該函式必須填入金鑰，AES的金鑰長度必須是16,24或32位元長度
2. 使用cipher.Block結構的方法NewGCM()來產生一個使用GCM加密模式的區塊加密物件(cipher.AEAD介面)
3. 最後，呼叫cipher.AEAD的Seal()方式來加密資訊；Seal()的函式特徵如下：

- Seal(dst, nonce, plaintext, additionalData []byte) []byte
  - plaintext是待加密的明文, nonce(為number only used one的簡寫)參數是個在加密/解密過程中只會使用一次的隨機數(以[]byte切片形式儲存), 好防堵重送攻擊(replay attack, 藉由攔截和送出同樣的通訊來竊取結果); 也就是說, 接收者必須用金鑰以及nonce來解密密文
  - dst參數是一個[]byte切片, 加密後的結果會附加到這個切片結尾; 下面我們利用了這個特性, 把nonce跟密文合併成單一一個字串傳給接收者, 後者會在解密時將之重新分開 (當然你也可以給dst填入nil, 以便將nonce獨立傳回給使用者)

- `additionalData` 參數 (additional authenticated data, 額外驗證資訊) 可以用來加入更多驗證資訊, 例如發送者電腦的MAC網路位址, 不過在此我們不使用它(填入nil)
- 解密過程的前兩個步驟與加密相同, 但最後要呼叫 `cipher.AEAD` 的 `Open()` 方法解密, 其參數和 `Seal()` 很像:

`Open(dst, nonce, ciphertext, additionalData []byte)([]byte, error)`

# 練習：AES對稱加密法

```
1  package main
2
3  import (
4      "crypto/aes"
5      "crypto/cipher"
6      "crypto/rand"
7      "fmt"
8      "os"
9  )
10
```

```
11 //加密函式
12 func encrypt(data, key []byte) (resp []byte, err error) {
13     //建立區塊加密物件
14     block, err := aes.NewCipher([]byte(key))
15     if err != nil {
16         return resp, err
17     }
18     //使用GCM加密模式
19     gcm, err := cipher.NewGCM(block)
20     if err != nil {
21         return resp, err
22     }
23     //產生一個gcm.NonceSize()長度的[]byte切片
24     nonce := make([]byte, gcm.NonceSize())
25     //用crypto/rand套件產生一個安全隨機數作為nonce
26     if _, err := rand.Read(nonce); err != nil {
27         return resp, err
28     }
29     //加密資料，並將結果附加到nonce尾端(傳回nonce + 密文)
30     resp = gcm.Seal(nonce, nonce, data, nil)
31     return resp, nil
32 }
33
```

```
34 //解密函式
35 func decrypt(data, key []byte) (resp []byte, err error) {
36     //和加密函式一樣，建立區塊加密物件並使用GCM加密模式
37     block, err := aes.NewCipher([]byte(key))
38     if err != nil {
39         return resp, err
40     }
41     gcm, err := cipher.NewGCM(block)
42     if err != nil {
43         return resp, err
44     }
45     //分割nonce加密文
46     nonce := data[:gcm.NonceSize()]
47     encryptedData := data[gcm.NonceSize():]
48     //解密資料(dst傳入nil; 若傳入[]byte切片，傳回結果就是dst + 解密字串)
49     resp, err = gcm.Open(nil, nonce, encryptedData, nil)
50     if err != nil {
51         return resp, fmt.Errorf("解密錯誤: %v", err)
52     }
53     return resp, nil
54 }
55
```



```
56 func main() {
57     data := "My secret text"
58     fmt.Printf("原始資料: %s\n", data)
59
60     //產生一個16位元長度隨機金鑰
61     key := make([]byte, 16)
62     if _, err := rand.Read(key); err != nil {
63         fmt.Println(err)
64         os.Exit(1)
65     }
66
67     //加密
68     encrypted, err := encrypt([]byte(data), key)
69     if err != nil {
70         fmt.Println(err)
71         os.Exit(1)
72     }
73     fmt.Printf("加密資料: %x\n", string(encrypted))
74
75     //解密
76     decrypted, err := decrypt(encrypted, key)
77     if err != nil {
78         fmt.Println(err)
79         os.Exit(1)
80     }
81     fmt.Printf("解密資料: %s\n", string(decrypted))
82 }
83
```

# 執行結果

```
PS D:\git\GO\ch18\18-3-1> go run .
```

```
原始資料: My secret text
```

```
加密資料: e9767f0e342609bf459d64d7eaf25ebeae5ce2f37c33de61fd2ca60d39bdf5780fa2e5ac68eaba64a82f
```

```
解密資料: My secret text
```

- 註：你看到的加密結果應該每次都不同

# 使用crypto/rand產生安全的隨機數

- 如在第一章提過，math/rand的隨機數是“偽隨機數”，在相同的亂數種子下會重複；但上面練習使用的crypto/rand套件來產生隨機性更強，可用於加密安全用途的亂數：

- `func Int(rand io.Reader, max *big.Int) (n *big.Int, err error)`

`rand.Int()` (不要和math/rand的`rand.Int()`混淆)接收一個形別為`big.Int` (見第三章大數值)的引數`max`, 然後會傳回0至`max-1`之間的隨機數，同樣是`big.int`形別：

```
package main
```

```
import (  
    "crypto/rand"  
    "fmt"  
    "math/big"  
)
```

```
func main() {  
    //產生0-999之間的亂數  
    r, _ := rand.Int(rand.Reader, big.NewInt(1000))  
    fmt.Println(r)  
}
```

## 18-3-2 非對稱加密法

- 非對稱加密法(**asymmetric encryption**)又稱公鑰加密法：它使用一組公鑰(**public key**)及私鑰(**private key**)，公鑰會自由分享給想跟你交換訊息的人，私鑰則由你私下持有
- 若有人想傳加密訊息給你，他們就可以用公鑰加密之，而這個訊息只能有你的私鑰才解的開；反過來說，若你想證明一段訊息確實出自你之手，你可以用私鑰加密訊息，那其他人只要用你的公鑰解開，就能驗證訊息的真實性 (稍後會看到何位數位簽章)

- Go語言同樣支援幾種常見的非對稱式加密演算法，如RSA(Rivest-Shamir-Adleman)和DSA(數位簽章演算法)
- 下面我們使用的是RSA-OAEP, OAEP即“最優非對稱加密填充”，是一種很常搭配RSA的演算法；這都可以在go語言內建的crypto/rsa套見找到
- Go語言中操作RSA-OAEP的步驟如下：

1. 使用`rsa.GenerateKey()`產生密鑰及其搭配的公鑰(`rsa.PrivateKey`指標結構)
2. 使用`rsa.EncryptOAEP()`對明文加密，此函式特徵如下：

```
func EncryptOAEP(hash hash.Hash, random io.Reader, pub *PublicKey, msg []byte, label []byte) ([]byte, error)
```

- 第一個參數`hash`是雜湊函式，用來扮演“隨機預言機” (random oracle)，以便產生均勻的真實隨機數，官方推薦使用`sha256`；第二個函式`random`用來提供亂數，一般會使用`crypto/rand`的`Reader`函式，好讓RSA對同一明文加密後不會得到相同結果
- `pub`是公鑰，它其實是來自前面的`rsa.PrivateKey`指標結構中；`msg`是待加密的明文，至於`label`則是需要附加到加密後訊息中的額外明文資訊，在此我們不使用

3. 解密時則使用`rsa.DecryptOAEP()`函式, 其參數和加密很像, 但必須使用私鑰來解密:

```
func DecryptOAEP(hash hash.Hash, random io.Reader, priv *PrivateKey, ciphertext []byte, label []byte) ([]byte, error)
```



# 練習：RSA-OAEP非對稱式加密法

```
1  package main
2
3  ∨ import (
4      "crypto/rand"
5      "crypto/rsa"
6      "crypto/sha256"
7      "fmt"
8      "os"
9  )
10
11 ∨ func main() {
12     data := []byte("My secret text")
13     fmt.Printf("原始資料: %s\n", data)
14
15     //產生私鑰(及公鑰)，長度2048位元
16     privateKey, err := rsa.GenerateKey(rand.Reader, 2048)
17     ∨ if err != nil {
18         fmt.Printf("產生私鑰錯誤: %v", err)
19         os.Exit(1)
20     }
21     publicKey := privateKey.PublicKey //公鑰就在PrivateKey結構中
22 }
```

```
23 //加密,使用SHA256,crypto/rand.Reader及公鑰
24 encrypted, err := rsa.EncryptOAEP(
25     sha256.New(), rand.Reader, &publicKey, data, nil)
26 if err != nil {
27     fmt.Printf("加密錯誤: %v", err)
28     os.Exit(1)
29 }
30 fmt.Printf("加密資料: %x\n", string(encrypted))
31
32 //解密, 使用SHA256,crypto/rand.Reader及私鑰<
33 decrypted, err := rsa.DecryptOAEP(
34     sha256.New(), rand.Reader, privateKey, encrypted, nil)
35 if err != nil {
36     fmt.Printf("解密錯誤: %v", err)
37     os.Exit(1)
38 }
39 fmt.Printf("解密資料: %s\n", string(decrypted))
40 }
41
```

執行結果如下：

```
PS D:\git\GO\ch18\18-3-2> go run .
```

```
原始資料: My secret text
```

```
加密資料: aafe6adba07809ff0a923390db55803e8376b838d0be54860c42cb3c53b6e2e86e8001d3d69b4f35b29a52203507e4935a33d8d9cb  
cdc4f4943f853121fe7d774379c7da341ca5db1ad2360a0460f8693ffbd34a122879f9dc55109f00fc460ddac1ac03b6e78dc880097dfb55c7cc  
51232fc1fa0fa0b644cf1a506b87f4781b53adce51930543d87460f2ea0824ddaaa3db2686de9e3d975c0a4980a5fbe7d83e11a919456934ac83  
9ebf6c906affb478689e8e8ad12d0d76f84b2d7377bbcf2262735d95257e1f78b0974ab491fba100bd7f5f0cc082d8d25939f5ac421c8d16207b  
bef0eea5f6f2290150c04a86fb22dc711e2d83c8c17cdc53be173b8b1f
```

```
解密資料: My secret text
```

## 18-4 數位簽章

- 數位簽章(digital signature)顧名思義就是數位簽名，它同樣運用了雜湊函式和公鑰加密法，但主要目的是驗證資料(比如文件,電郵)的完整性和傳送者身分，其運作方式如下：
  1. 傳送者先用雜湊函式產生資料的摘要(digest,類似checksum)
  2. 傳送者用私鑰加密摘要，此及為該份資料的數位簽章
  3. 他人收到資料後，同樣用雜湊函式產生一份摘要，並用公鑰來解密數位簽章
- go語言提供了一個相當方便的數位簽章套件：`crypto/ed25519`；此套件使用了Ed25519數位簽章演算法，它會使用SHA256為雜湊函式

練習：使用數位簽章

```
1  package main
2
3  ▼ import (
4      "crypto/ed25519"
5      "crypto/rand"
6      "fmt"
7      "os"
8  )
9
10 ▼ func main() {
11     data := []byte("My secret document")
12
13     //產生公鑰與私鑰(使用crypto/rand產生亂數)
14     publicKey, privateKey, err := ed25519.GenerateKey(rand.Reader)
15     ▼ if err != nil {
16         fmt.Println(err)
17         os.Exit(1)
18     }
19
20     //用私鑰產生數位簽章
21     signedData := ed25519.Sign(privateKey, data)
22     fmt.Printf("數位簽章:\n%x\n", signedData)
23
24     //用公鑰，資料和數位簽章來驗證簽章是否有效
25     verified := ed25519.Verify(publicKey, data, signedData)
26     fmt.Println("驗證:", verified)
27 }
28
```

# 執行結果：

```
PS D:\git\GO\ch18\18-4> go run .
```

數位簽章：

```
d67499fb83f10b8ea78c86c41e429773e40535f014c0f83a05a44dc36c1d897a27156f8c6698b17501a  
7bbc468f4cfbb69f94f10ef5b8e5cd7d7b31e617c440a
```

驗證：true

## 18-5 HTTPS/TLS與X.509憑證

- TLS(傳輸層安全系協定)能確保資訊在傳遞過程中的安全
- 這個協定能確保以下項目：
  - 讓客戶端和伺服器都使用數位簽證(digital certificates)來代表身分
  - 可要求客戶端和伺服器使用公鑰加密法來驗證身分
  - 產生訊息摘要(digest), 確保資料在傳送過程中不受竄改
  - 訊息在傳送時也會加密, 使其對第三方保持機密



- 在使用TLS時，客戶端與伺服器會展開TLS交握協議(TLS handshake), 傳送憑證給彼此，並在身分獲得驗證(憑證是受信任的)後交換對話金鑰(session key, 例如對稱式加密的金鑰), 以便安全地交換訊息，確保資料在傳輸過程安全無虞；TLS最廣泛使用的公鑰憑證標準之一是X.509憑證
- 基本上，一個憑證會包括使用者身分/位置(網址)/憑證公鑰/有效日期/數位簽章等，用來證明公鑰由該用者擁有
- 該憑證若有效(能夠被信任), 就得由一個受信任的CA(Certificate Authority, 憑證授權中心)簽署之；使用者向CA提出申請後，CA會以自己的私鑰對使用者的公鑰產生數位簽章，於是其他人只要用該CA的公鑰解開數位簽章，就能知道使用者的身分及公鑰是否有效 (在許多地方，有效的憑證是具備法律效力的)

- 當然, 你也可以用自己的私鑰簽署自己的憑證(自己兼任CA), 這就是“自簽署憑證(self-signed certificate)”; 這種憑證對外並無效力, 但你可以將憑證事先加入你的客戶端及伺服器, 讓他們透過HTTPS通訊時能夠信任彼此和使用加密通訊
- 下面我們就來看到如何在客戶端與伺服器間產生使用TLS和X.509數位簽章, 這在go語言透過crypto/tls以及crypto/x509套件來實現

# 練習：產生自簽署憑證並用於客戶端/伺服器

- 這個練習會包括三個程式檔：用來產生私鑰及憑證的`cert.go`(它產生的`.pem`與`.key`檔案會存放於專案根目錄下)，以及客戶端`client.go`和伺服器`server.go`, 位於各自的子資料夾中
- 此外`server_simple.go`是個簡單版的HTTPS/TLS伺服器，我們也會用它來展示TLS在伺服器在瀏覽器的運作效果

18-5\

cert\

cert.go

client\

client.go

server\

server.go

server\_simple\

server\_simple.go

Client\_cer.pem      客戶端憑證

Client.key            客戶端私鑰

Server\_cert.pem      伺服器憑證

Server.key            伺服器私鑰

# 憑證與私鑰產生程式：cert.go

```
1  package main
2
3  import (
4      "crypto/rand"
5      "crypto/rsa"
6      "crypto/x509"
7      "crypto/x509/pkix"
8      "encoding/pem"
9      "log"
10     "math/big"
11     "net"
12     "os"
13     "time"
14 )
15
16 const (
17     clientCertName = `.\client_cert.pem`
18     clientKeyName  = `.\client.key`
19     serverCertName = `.\server_cert.pem`
20     serverKeyName  = `.\server.key`
21     host           = "127.0.0.1"
22     hostDNS        = "localhost"
23 )
24
```

```
25  ✓ func main() {
26      if err := generateCert(clientCertName, clientKeyName); err != nil {
27          log.Println(err)
28      }
29      log.Println("產生:", clientCertName, clientKeyName)
30
31      if err := generateCert(serverCertName, serverKeyName); err != nil {
32          log.Println(err)
33      }
34      log.Println("產生:", serverCertName, serverKeyName)
35  }
36
37  //產生憑證的函式
38  func generateCert(certFile, keyFile string) error {
39      //產生一個安全的隨機樹當作序號
40      serialNumber, err := rand.Int(rand.Reader, big.NewInt(1000))
41      if err != nil {
42          return err
43      }
44
45      now := time.Now() //取得現在時間
46  }
```

```
47 //產生X.509憑證
48 ca := &x509.Certificate{
49     //持有人資訊
50     Subject: pkix.Name{
51         CommonName:    "Company",
52         Organization:    []string{"Company, INC."},
53         Country:          []string{"US"},
54         Province:         []string{"",
55         Locality:         []string{"San Francisco"},
56         StreetAddress:    []string{"Golden Gate Bridge"},
57         PostalCode:       []string{"94016"},
58     },
59     //序號
60     SerialNumber:      serialNumber,
61     //簽章加密法
62     SignatureAlgorithm: x509.SHA256WithRSA,
63     //生效時間(即現在時間)
64     NotBefore:          now,
65     //有效時間(現在開始2年後)
66     NotAfter:           now.AddDate(2, 0, 0),
67     //公鑰用途(可用來簽署憑證, 要用於數位簽證)
68     KeyUsage:           x509.KeyUsageCertSign | x509.KeyUsageDigitalSignature,
69     //公鑰額外用途(客戶端驗證/伺服器驗證)
70     ExtKeyUsage:         []x509.ExtKeyUsage{x509.ExtKeyUsageClientAuth, x509.ExtKeyUsageServerAuth},
71     BasicConstraintsValid: true,
72     //憑證可當CA使用
73     IPAddresses:         []net.IP{net.ParseIP(host)},
74     //憑證可使用的網址與網域
75     DNSNames:            []string{hostDNS},
76 }
77
```

```
78 //用RSA產生私鑰
79 privateKey, err := rsa.GenerateKey(rand.Reader, 2048)
80 if err != nil {
81     return err
82 }
83
84 //以憑證/私鑰和其公鑰來簽署該憑證
85 //x509.CreateCertificate()的第二參數是待簽署的憑證
86 //第三個參數則是CA的憑證；兩者相同代表是自簽署憑證
87 //傳回值DER為憑證內容，是[]byte切片
88 DER, err := x509.CreateCertificate(
89     rand.Reader, ca, ca, &privateKey.PublicKey, privateKey)
90 if err != nil {
91     return err
92 }
93
94 //將憑證字串轉成PEM(Privacy Enhanced Mail, Base64編碼)格式
95 cert := pem.EncodeToMemory(
96     &pem.Block{
97         Type: "CERTIFICATE",
98         Bytes: DER,
99     })
100
101 //將私鑰轉成PEM格式
102 key := pem.EncodeToMemory(
103     &pem.Block{
104         Type: "RSA PRIVATE KEY",
105         Bytes: x509.MarshalPKCS1PrivateKey(privateKey),
106     })
```



```
107
108 //將憑證與私鑰(私鑰只限擁有者存取)存為檔案
109 //憑證權限設為0777(可由任何人自由存取)
110 if err := os.WriteFile(certFile, cert, 0777); err != nil {
111     return err
112 }
113 //私鑰權限設為0600(只有擁有者能讀寫)
114 if err := os.WriteFile(keyFile, key, 0600); err != nil {
115     return err
116 }
117
118 return nil
119 }
120
```

執行結果：


```
PS D:\git\GO\ch18\18-5> go run .\cert\cert.go  
2022/09/25 21:20:49 產生: .\client_cert.pem .\client.key  
2022/09/25 21:20:49 產生: .\server_cert.pem .\server.key
```

- 這會在ch18/18-5根目錄下面產生2個憑證及2個私鑰檔
- 伺服器得到的憑證檔內容會類似如下：

ch18 > 18-5 >  server\_cert.pem

```
1  -----BEGIN  CERTIFICATE-----
2  MIID5zCCAs+gAwIBAgICAw8wDQYJKoZIhvcNAQELBQAwwYcxCzAJBgNVBAYTA1VT
3  MQkwBwYDVQQIEwAxXjAUBgNVBAcTDVNhbiBGcmFuY2lzY28xGzAZBgNVBAkTEkdv
4  bGR1biBHYXR1IEJyaWRnZTEOMAwGA1UEERMFOTQwMTYxXjAUBgNVBAoTDUNvbXBh
5  bnksIEl0Qy4xEDA0BgNVBAMTB0NvbXBhbnkwHhcNMjIwOTI1MTMyMDQ5WhcNMzIw
6  OTI1MTMyMDQ5WjCBhzELMAkGA1UEBhMCVVMxCTAHBgNVBAGTADEWMBQGA1UEBxMN
7  U2FuIEZyYW5jaXNjbzEbMBkGA1UECRMSR29sZGVuIEdhGUGnJpZGdlMQ4wDAYD
8  VQQREwU5NDAXNjEWMBAQGA1UEChMNQ29tcGFueSwSU5DLjEQA4GA1UEAxMHQ29t
9  cGFueTCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAL6pvf0WGIW2B44
10  21m+wPMillCJJbSAFW1cyg+PYBCCG+Gd0IDq/K/WSf4CnAC4dbcIHUGCeCf5kA/u
11  HfFKh1k+arTjy5USRHSfiq5Gch7CpOMsFWbZ9RS+YteYW6R7Pkg27ohGJ3v6cW1k
12  g86CO+HqdHKARNeXSEpfNkeVBs91p8P5oMhaXnZFaR38Nbx8WEYmo0/Lsx74xdVA
13  rPQ0mVVq6rVA1opfpJLVuPuWV/dMf4UNBuQ+NfiAw8au9rTmEYeFU2bw+5NF53su
14  T46FxKXiyVpffNA1e2JrWtYNpliTbouewaNugsXHnXTloG6cKcsD01a+NZrB9ddn
15  Bq7lQ20CAwEAANbMFkwDgYDVROPAQH/BAQDAgKEMB0GA1UdJQQWMBQGCCsGAQUF
16  BwMCBggrBgEFBQcDATAMBGNVHRMBAf8EAjAAMBoGA1UdEQQTMBGCCWxvY2FsaG9z
17  dIcEfWAAATANBgkqhkiG9w0BAQsFAAOCAQEAnwVGHg0+tn/aeg6Opu/ksU1JyrxC
18  lrJK5naJ5lCgw8gwRbJ9f88zPEZFiiGcjgxgW7UjuWpjy2f5tCzHo9IEADeL3TcO
19  /UV/YCeS07RWDbafUAL/XTuSW0IAaHViDiQT5sSENx0TaZ6GJlmTMnM1m5L170Uz
20  du9y/nH7AMW4R6+jWS8VAhMkK9UW2Cs6RIexoie1qJDwFQRbKGs208YDfXZV/v7W
21  YN/A7XGpMTW1TMh1Ptg/g84V6M6gk5NLqopeL/nxXgfc/wy1sMDWo1a2D+ZI04Hv
22  9+tn8KI5XRE8fcSc4x9Twbl69AIbA8fuK9N22yRK3BwBLiAKFHs1S/QK+Q==
23  -----END  CERTIFICATE-----
24
```

- 至於以下則是伺服器的私鑰內容：

ch18 > 18-5 >  server.key

```
1 -----BEGIN RSA PRIVATE KEY-----
2 MIIEpAIBAAKCAQEAvm9/RYYibDYHjjbWb7A8yKWUIkltIAVbVzKD49gEIIb4Z04
3 gOr8r9ZJ/gKcALh1twgdQYJ4J/mQD+4d8UqGWT5qtOPL1RJEdIWkrkZyHsKk4ywV
4 Ztn1FL5i15hbpHs+SDbuiEYne/pxbWSDzoI74ep0coBE15dISl82R5UGz3Wnw/mg
5 yFpedkVpHfw1vHxYRiajT8uzHvjF1UCs9DSZVWrqtUCWil+kktW4+5ZX90x/hQ0G
6 5D41+IDDxq72tOYRh4VTZvD7k0Xney5PjoXEpeLJWl980DV7Ymta1g2mWJMGi57B
7 o26Cxcedd0WgbpwpypwPTVr41msH112cGruVDbQIDAQBAoIBAAAnqKccPbxc56wMn
8 UcDI+p+qHmbzWtKYFoUpXhiRDB3PpkOFOXr6vUkni5F58ULYVMa/1Uwi0zzA7Yzm
9 hQgXKAVcCZB+iAeoe4bTqvKFF0oiNRDbKgG/M89wVxN/CMT047g9ownTHBqGW3+k
10 ynXir9p/pq6NnQbZBx1w+iuexEv1azs/KHJ1hZ0g3hVR7yz8zdRhgm70OE4OpqYF
11 nBR5U1ZAAY7kASF/GqrryG+U5H2VNgbP91sXIxhoo8JcQNRs6FFKKVcGMZppvMW9
12 qUB8mDchB+jHs3zOXR+uPlgV/43pR30quL9gJ95AN9bv2mTCVaageyJqZAxT+q/R
13 WyrQhMECgYEA+oASx2mr9iJzsEEo5ASuTTd99W2pol4wpkoCjJ1s77K0eWuxrOLR
14 VoTp25NJrFFPCUb4TYCH4H7ZAqj8m4v1HQkG7qdKqN9mNeBCzUK9/Ka1jzVN5Apu
15 RsAZoFc5OpFcoyGvLjQt4qBlnkG/7oJU3qSTS3dppmk1vKC1DD2cumMCgYEAwt1b
16 J17Vdo0Fwzb3Cakx1mLXCmD+E2+7wD6ag1Zfyh56gP6fvMWVzeZ1hrPrAzMVOnKD
17 bz6RNkKwdJJHJjnv/3JJ6hFp1MK1dsCYp7GAN33hlMAj58IIkB+pnSzGoF38S5sM
18 sLkZnxBX7KJYEu255FE/HxdnTcm1T80cuKB7C+8CgYEAsOzkVpP8MMwEieh7yRVc
19 jo0zGbVqqpN8KPb89fP6jRHAl2Ix7rnwAQaAGWPE3YaLKNDnPm6/oSZIZfZUsN3
20 TBqkGsttn/ipaEQM6ozJQzk74vnzGa2EVdQ4RVdVxFgG5fFUmX2hKv++xhgKR5s1
21 9lqm7hZZOH/rd17KgOrDV6ECgYA1ErVEfQ1R1Em8ia0yYXaTSurd96CctLOEQskd
22 expuWGzv0+s4p00P/3UFst4RqglfOS/ZzkYJbJLZvbpJjEB16PB/JC0JCxe8x+sM
23 ykl1tVA1s1gUxrGVetHEj0b1skw0UnvAO9uwmqnH4j6PDzCL1MLgrxNkr1ABGQLbg
24 hz3c2wKBgQDhM1Yfna1yy1MYTZYFeCK+jc+x3yuIxPDyKX/GewPLIcHyTi7Nr/Hu
25 RTEcwV91A8ENF2M0tyJP1tCS0q0HJbdwgJHW9f1Qr/r/xM8iPTykj4A12Lgi4+lZ
26 w+OGkQDNLM6qFw1JSN35Bo/h64oquydDzOxphJKYzB05tA58/dMHBg==
27 -----END RSA PRIVATE KEY-----
```

# 簡易HTTPS/TLS示範：`server_simple.go`

- 在正式看來伺服器要如何應付客戶端的憑證之前，我們先來看個簡單的HTTPS/TLS範例：

```
1  package main
2
3  ∨ import (
4      |     "log"
5      |     "net/http"
6      | )
7
8  ∨ const ( //憑證及私鑰檔名
9      |     serverCertName = `.\server_cert.pem`
10     |     serverKeyName  = `.\server.key`
11     | )
12
13  ∨ func main() {
14      |     log.Println("啟動伺服器")
15      |     //對路徑指定請求處理函式
16      |     http.HandleFunc("/", hello)
17      |     //啟動HTTPS/TLS伺服器，仔入憑證和私鑰
18      |     log.Fatal(http.ListenAndServeTLS(":8080", serverCertName, serverKeyName, nil))
19      | }
20
21     //HTTP請求處理函式
22  ∨ func hello(w http.ResponseWriter, r *http.Request) {
23      |     log.Println("收到請求")
24      |     w.Write([]byte("Hello Golang from a secure server"))
25      | }
26
```



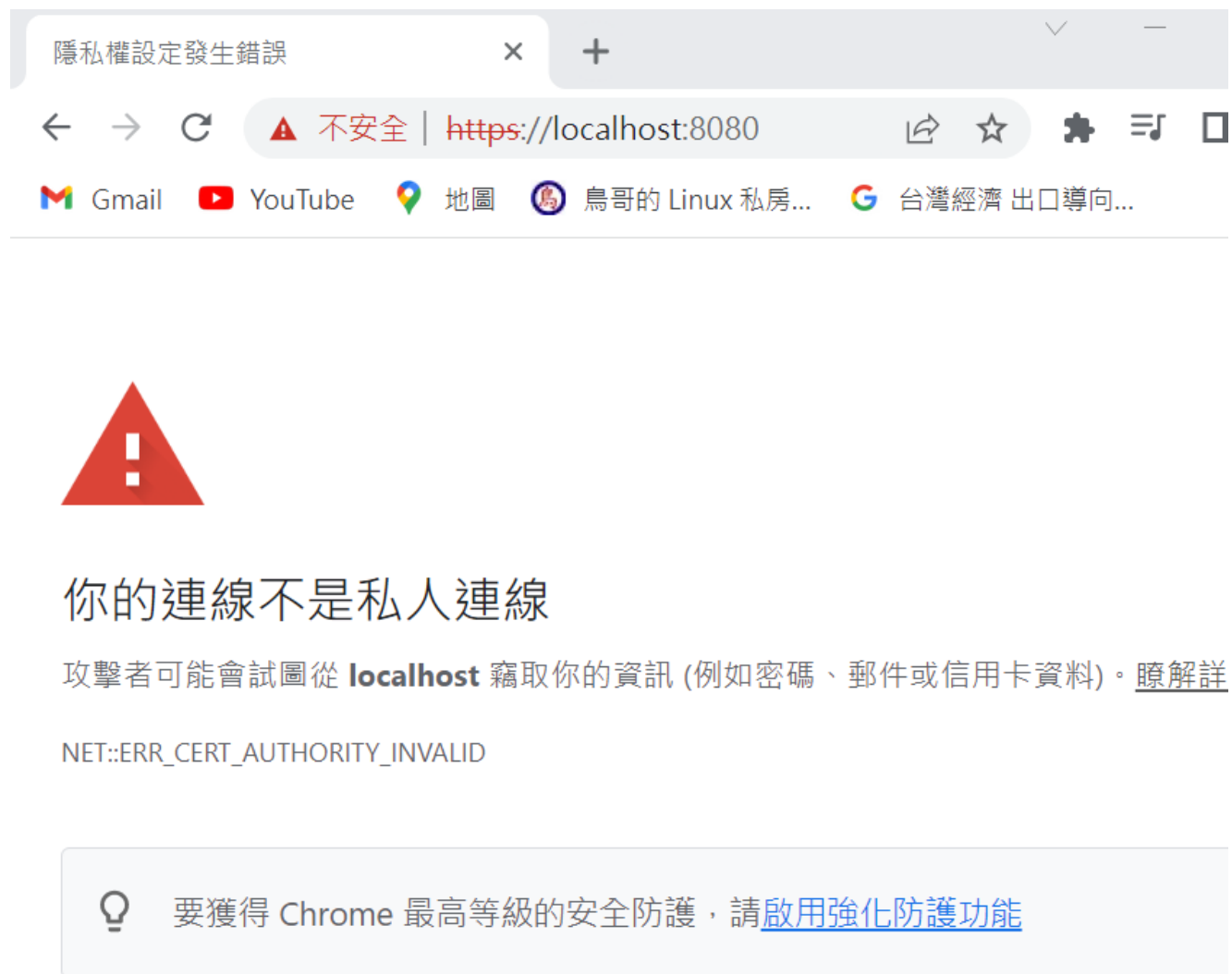
- 現在來試著執行這個伺服器：

```
PS D:\git\GO\ch18\18-5> go run .\server_simple\server_simple.go
```

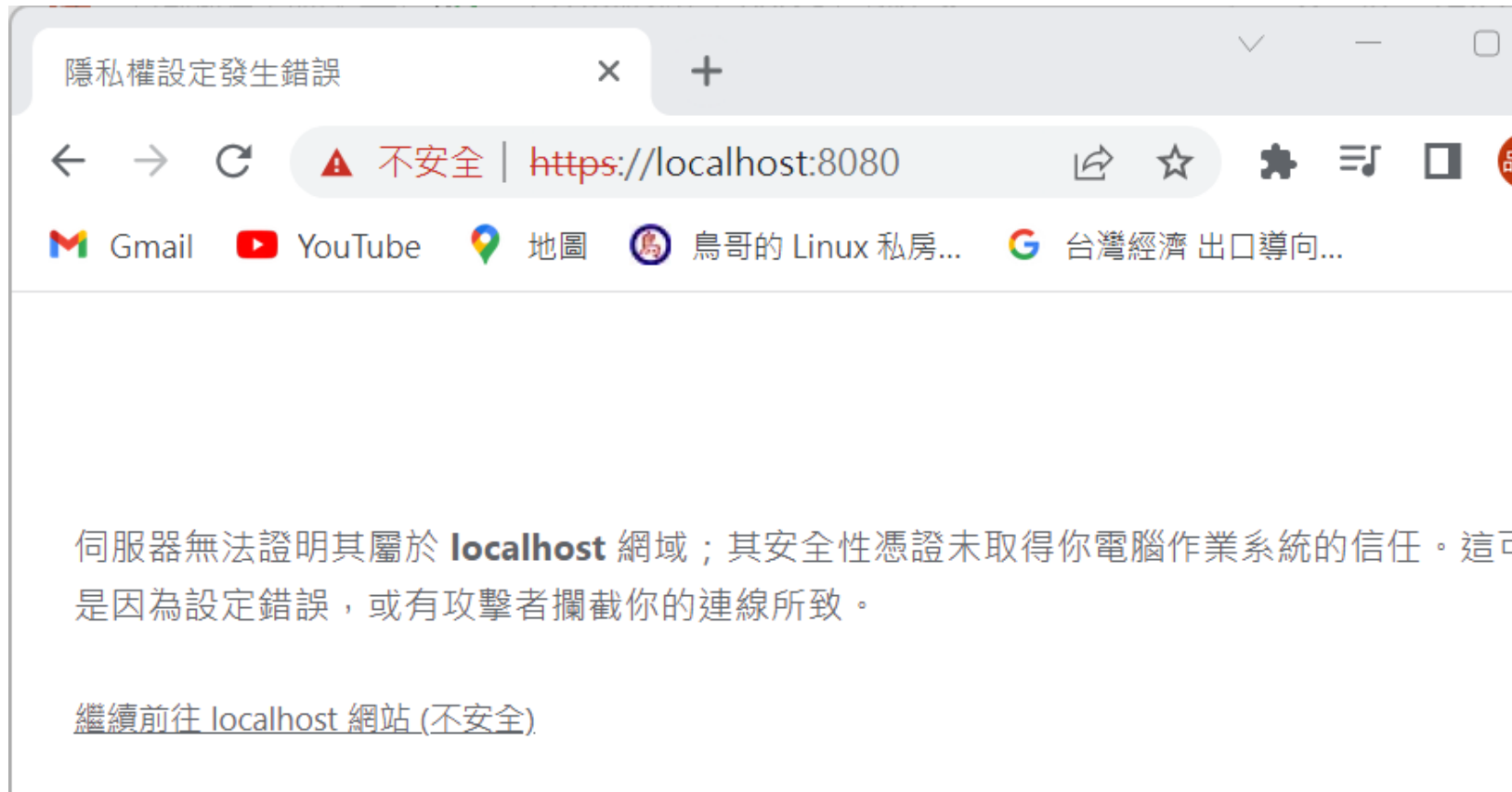
```
2022/09/26 22:24:47 啟動伺服器
```

```
█
```

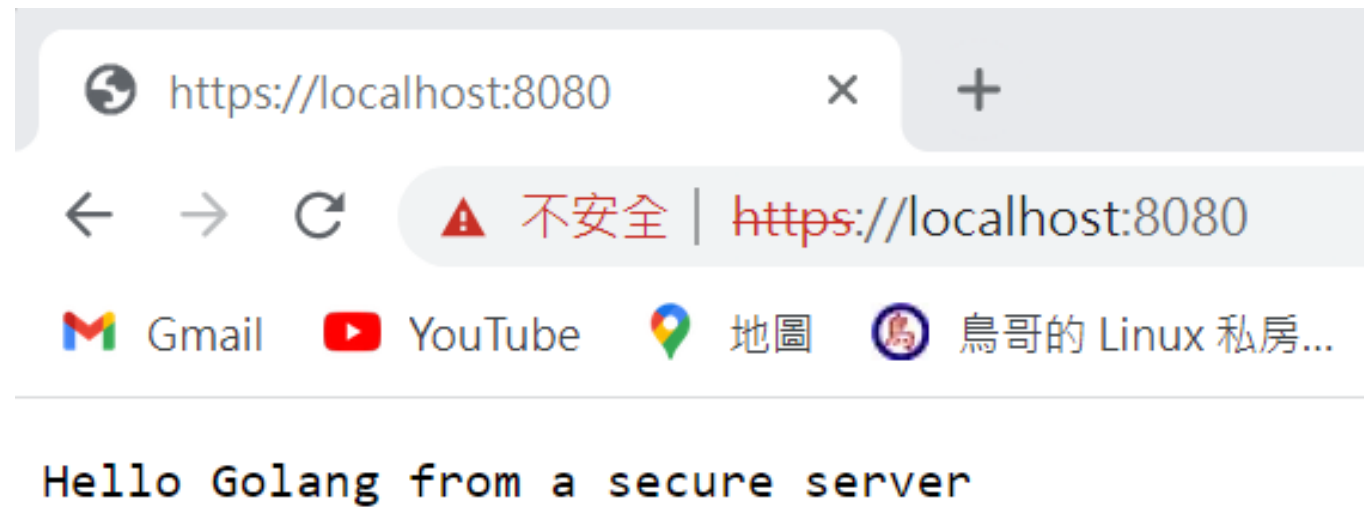
- 然後在網頁瀏覽器輸入 <https://localhost:8080>



- 出現以上畫面是正常的，因為瀏覽器無法信任這個伺服器的憑證 (既然是自簽署的，當然不在瀏覽器信任的CA清單中)
- 這時我們還是可以按“進階”：



- 再點“繼續前往localhost網站(不安全)”，就能看到伺服器程式的回應：



- 看看server\_simple.go在主控台的輸出資訊，也證明了TLS發揮了作用：

```
PS D:\git\GO\ch18\18-5> go run .\server_simple\server_simple.go
2022/09/26 22:24:47 啟動伺服器
2022/09/26 22:26:07 http: TLS handshake error from [::1]:49172: remote error: tls:
unknown certificate 瀏覽器拒絕TLS交握協議
2022/09/26 22:26:07 http: TLS handshake error from [::1]:49173: remote error: tls:
unknown certificate
2022/09/26 22:26:07 http: TLS handshake error from [::1]:49174: remote error: tls:
unknown certificate
2022/09/26 22:30:11 http: TLS handshake error from [::1]:49209: remote error: tls:
unknown certificate
2022/09/26 22:30:11 http: TLS handshake error from [::1]:49210: remote error: tls:
unknown certificate
2022/09/26 22:30:11 收到請求 使用者選擇連線, 因此收到請求
2022/09/26 22:30:11 收到請求
█
```

- 可見要在go語言建立HTTPS/TLS伺服器是非常容易的，你也可以讓伺服器改用真正的CA機構所簽署的有效憑證
- 但這樣仍有個問題，就是伺服器不會檢查客戶端的憑證，因此你還是可以連線和取得回應
- 下面我們來看個更複雜，更正式的例子：如何利用憑證來檢查客戶端的連線身分

# 伺服器程式：server.go

- 為了讓伺服器能驗證客戶端的憑證，你需要將客戶端憑證加入伺服器的**CA**清單，並要求伺服器驗證憑證
- 為此我們得使用**http.Server**結構來設定**TLS**要用的憑證

```
1  package main
2
3  import (
4      "crypto/tls"
5      "crypto/x509"
6      "log"
7      "net/http"
8      "os"
9  )
10
11  const (
12      clientCertName = `.\client_cert.pem`
13      serverCertName = `.\server_cert.pem`
14      serverKeyName   = `.\server.key`
15      host            = "localhost"
16      port            = "8080"
17  )
18
19  func main() {
20      //讀取客戶端憑證
21      clientCert, err := os.ReadFile(clientCertName)
22      if err != nil {
23          log.Fatal(err)
24      }
25  }
```



```
26 //取得系統的憑證存放區(CertPool)
27 clientCAs, err := x509.SystemCertPool()
28 if err != nil {
29     clientCAs = x509.NewCertPool()
30 }
31 //將PEM格式的客戶端憑證字串加入CertPool
32 if ok := clientCAs.AppendCertsFromPEM(clientCert); !ok {
33     log.Println("加入客戶端憑證錯誤")
34 }
35
36 //TLS設定
37 tlsConfig := &tls.Config{
38     ClientCAs:  clientCAs,
39     ClientAuth: tls.RequireAndVerifyClientCert,
40 }
41
42 //設定http.Server結構
43 server := &http.Server{
44     Addr:      host + ":" + port, //伺服器網址
45     Handler:   nil,                //用預設的DefaultServerMux結構來處理路徑
46     TLSConfig: tlsConfig,
47 }
48
```

```
49     log.Println("啟動伺服器")
50     http.HandleFunc("/", hello)
51     //啟動HTTPS/TLS伺服器並載入伺服器憑證/私鑰
52     log.Fatal(server.ListenAndServeTLS(serverCertName, serverKeyName))
53 }
54
55 func hello(w http.ResponseWriter, r *http.Request) {
56     log.Println("收到請求")
57     w.Write([]byte("Hello Golang from a secure server"))
58 }
59
```

- 伺服器程式會試著取得你系統上的憑證存放區，因此你其實也可以將客戶端的憑證安裝到系統中
- 在以上程式，`tls.Config`結構的參數`ClientAuth`用來指定伺服器要如何驗證客戶端的憑證，他可以設定為以下常數：
  - `tls.NoClientCert`：不要求也不驗證客戶端憑證(預設)
  - `tls.RequestClientCert`：要求但不強制提供客戶端憑證
  - `tls.RequireAnyClientCert`：要求提供至少一份客戶端憑證
  - `tls.VerifyClientCertIfGiven`：要求但不強制提供客戶端憑證，若有提供則會驗證
  - `tls.RequireAndVerifyClientCert`：要求提供至少一份客戶端憑證並驗證

# 客戶端程式：client.go

- 對於客戶端程式，他則同樣得將伺服器的憑證加入自己的信任CA列表，才能讓TLS交握協議順利通過

```
1  package main
2
3  ∨ import (
4      "crypto/tls"
5      "crypto/x509"
6      "io"
7      "log"
8      "net/http"
9      "os"
10 )
11
12 ∨ const (
13     clientCertName = `.\client_cert.pem`
14     clientKeyName  = `.\client.key`
15     serverCertName = `.\server_cert.pem`
16     host           = "localhost"
17     port           = "8080"
18 )
19
20 ∨ func main() {
21     ∨ //載入客戶端憑證及私鑰，產生成tls.Certificate物件
22       //以便放入後面的TLS設定中
23     cert, err := tls.LoadX509KeyPair(clientCertName, clientKeyName)
24     ∨ if err != nil {
25         log.Fatal(err)
26     }
27 }
```

```
28 //讀取伺服器憑證
29 serverCert, err := os.ReadFile(serverCertName)
30 if err != nil {
31     log.Fatal(err)
32 }
33
34 //取得系統憑證存放區或新建一個CertPool
35 rootCAs, err := x509.SystemCertPool()
36 if err != nil {
37     rootCAs = x509.NewCertPool()
38 }
39
40 //將PEM格式的伺服器加入CertPool
41 if ok := rootCAs.AppendCertsFromPEM(serverCert); !ok {
42     log.Fatal("加入伺服器憑證錯誤")
43 }
44
45 //TLS設定，放入客戶端憑證以及信任的伺服器CA清單
46 tlsConfig := &tls.Config{
47     Certificates: []tls.Certificate{cert},
48     RootCAs:      rootCAs,
49 }
50
```

```
51 //建立http.Client結構，設定其傳輸層參數使用前面的TLS設定
52 client := &http.Client{
53     Transport: &http.Transport{
54         |     TLSClientConfig: tlsConfig,
55         |     },
56     }
57
58 //客戶端送出請求
59 resp, err := client.Get("https://" + host + ":" + port)
60 if err != nil {
61     |     log.Fatal(err)
62 }
63 defer resp.Body.Close()
64
65 //讀取伺服器回應
66 data, err := io.ReadAll(resp.Body)
67 if err != nil {
68     |     log.Fatal(err)
69 }
70
71 log.Println("收到回應:", string(data))
72 }
73
```

- 可以看到加入憑證到CA清單的過程和伺服器很像，只不過要使用 `tls.LoadX509KeyPair()` 來載入客戶端自身的憑證和私鑰 (此函式也可用在伺服器端，若是這樣的話 `ListenAndServeTLS()` 的參數填入空字串即可)
- 補充：若在讀取憑證與私鑰時，你的程式中仍保存著這兩份資料的PEM格式字串，那你可以用 `tls.X509KeyPair()` 函式直接把他們轉程 `tls.Certificate` 物件：

```
func X509KeyPair(certPEMBlock, keyPEMBlock []byte) (Certificate, error)
```



# 執行HTTPS/TLS練習

- 現在所有程式都寫好，客戶端與伺服器的憑證/私鑰檔也都產生好了，我們就可以來執行這個練習，看看雙方的憑證是否能用於TLS交握協議：

- 首先啟動伺服器：

```
PS D:\git\GO\ch18\18-5> go run .\server\server.go
2022/09/26 23:12:59 啟動伺服器
█
```

- 接著在新的主控台執行客戶端：

```
PS D:\git\GO\ch18\18-5> go run .\client\client.go
2022/09/26 23:14:23 收到回應: Hello Golang from a secure server
```

- 可見客戶端從伺服器得到了回應，會去看伺服器主控台，可以看到：

```
PS D:\git\GO\ch18\18-5> go run .\server\server.go
2022/09/26 23:12:59 啟動伺服器
2022/09/26 23:14:23 收到請求
█
```

- 順帶一提，本練習的客戶端與伺服器都可以透過https:localhost:8080和https:127.0.0.1:8080作為連線或監聽服務的網址，這兩者都已在憑證中，所以不會有問題

# 改用ECDSA簽章演算法

- 除了用RSA, 也可以用ECDSA(橢圓曲線數位簽章演算法), 這是一種處理速度比RSA更快, 私鑰比RSA更短, 但安全度相當的技術, 這實作於go內建套件crypto/ecdsa
- 你可以將cert/cert.go中X.509簽證使用的簽證演算法改成如下:

56



```
SignatureAlgorithm: x509.ECDSAWithSHA256,
```

- 然後產生私鑰並將之寫入檔案的過程則變成如下：

```
66      //產生ECDSA私鑰
67      privateKey, err := ecdsa.GenerateKey(elliptic.P521(), rand.Reader)
68      if err != nil {
69          return err
70      }
71
```

```
84      //將ECDSA私鑰轉成PEM格式
85      pemByte, err := x509.MarshalECPrivateKey(privateKey)
86      if err != nil {
87          return err
88      }
89      key := pem.EncodeToMemory(
90          &pem.Block{
91              Type: "PRIVATE KEY",
92              Bytes: pemByte,
93          })
94
```

- `ecdsa.GenerateKey()`函式的第一個參數`elliptic.P521()`來自`crypto/elliptic`套件，為ECDSA要使用的數學曲線，有`p224`,`p256`,`p384`,`p521`四種可選
- 本章結束