Guião de apoio 8
Comunicação Segura com SSL/TLS

---

1. Introdução ao tema

Uma das formas mais simples de concretizar comunicação segura em aplicações é utilizar o protocolo SSL/TLS (doravante vamos usar no texto TLS apenas). Este protocolo funciona sobre TCP/IP e faz uso de várias técnicas criptográficas para proteger a autenticação, a integridade e a confidencialidade dos dados transmitidos. A Figura 1 ilustra o protocolo.

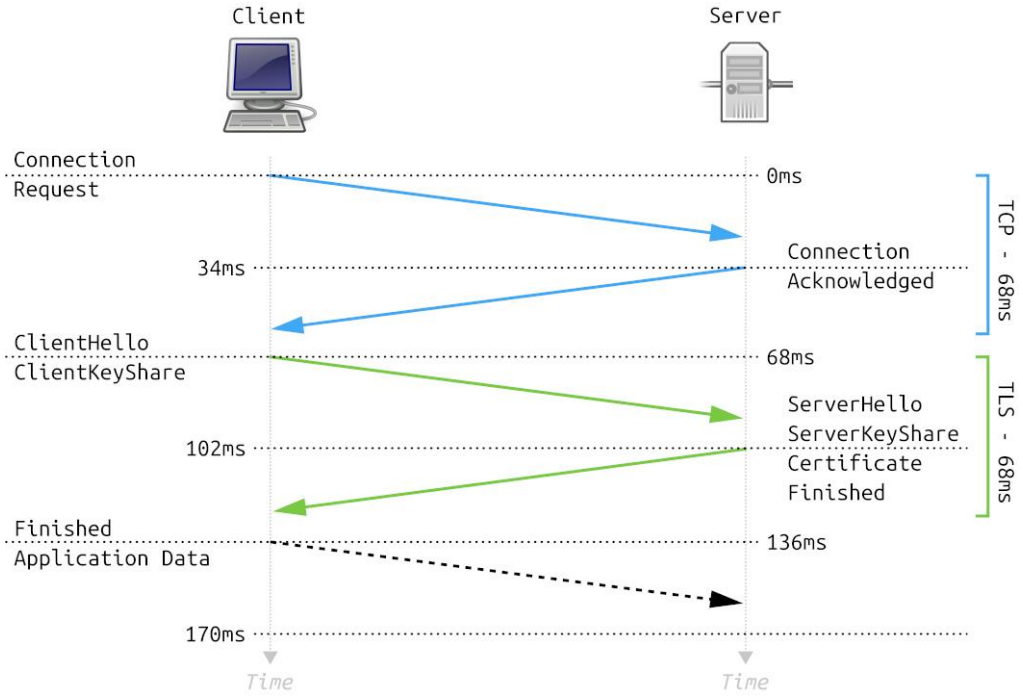

Figura 1. O protocolo SSL/TLS.

Não vamos discutir os detalhes do protocolo neste documento, mas recomendamos os alunos a reverem os conceitos relacionados com o protocolo nos slides da última aula TP. Neste guião, vamos nos centrar na concretização de comunicação segura numa aplicação distribuída em que clientes e servidores comunicam por TCP.

2. Criação de chaves e certificados assinados por uma CA

2.1. Para que os certificados do cliente e do servidor possam ser assinados por uma CA (Certificate Authority), é necessário criar um certificado para a CA fictícia. Crie primeiro a chave da CA:

```
$ openssl genrsa -out root.key 2048
```

2.2. Crie então um certificado auto-assinado para a CA:
(Deixe o campo `Common Name` em branco quando for pedido)

```
$ openssl req -x509 -new -nodes -key root.key -sha256 -days 365 -out root.pem
```

2.3. De seguida, serão geradas as chaves do servidor e do cliente de acordo com o seguinte padrão de comando:

```
$ openssl genrsa -out cli.key 2048
$ openssl genrsa -out serv.key 2048
```

2.4. Para que a CA assine os certificados do cliente e do servidor, emitir-se-á um pedido de assinatura de certificado para cada um. Isso pode ser feito de acordo com os seguintes comandos:
(Nestes comandos insira *localhost* no campo `Common Name`)

```
$ openssl req -new -nodes -key cli.key -sha256 -days 365 -out cli.csr
$ openssl req -new -nodes -key serv.key -sha256 -days 365 -out serv.csr
```

2.5 Na posse dos pedidos de assinatura, a CA pode proceder à geração dos certificados assinados por si. Isso pode ser feito para o cliente e para o servidor pelo seguinte padrão de comando:

```
$ openssl x509 -req -in cli.csr -CA root.pem -CAkey root.key \
-CAcreateserial -out cli.crt -days 365 -sha256
$ openssl x509 -req -in serv.csr -CA root.pem -CAkey root.key \
-CAcreateserial -out serv.crt -days 365 -sha256
```

Através deste método, o cliente e o servidor deverão no final, cada um, ter um par de ficheiros `<nome>.crt` e `<nome>.key`. Estes serão utilizados nos programas para que a autenticação e comunicação cifrada com o interlocutor sejam possíveis. Na implementação da autenticação, os programas cliente e servidor terão de utilizar o certificado da CA (`root.pem`) para validar a assinatura do certificado apresentado pelo interlocutor.

3. Uso de SSL/TLS no Python

Para transformar uma socket TCP numa SSL/TLS basta criar um contexto `SSLContext` do módulo `ssl` do Python [3] e utilizar a função `wrap_socket`. O contexto recebe um conjunto de informações relativas ao TLS (e.g., versão do protocolo, caminhos para ficheiros com chaves e certificados) e a função `wrap_socket` recebe como parâmetro a socket TCP previamente criada e já ligada.

3.1. Autenticação Unilateral (Servidor autentica-se para o cliente)

A seguir apresentamos trechos de códigos para concretizar clientes e servidores num cenário de autenticação unilateral. Neste caso, apenas o cliente verifica a identidade do servidor que comunica com qualquer cliente.

Cliente

```
import ssl
...
context = ssl.SSLContext(protocol=ssl.PROTOCOL_TLS_CLIENT)
context.verify_mode = ssl.CERT_REQUIRED
context.check_hostname = True
context.load_verify_locations(cafile='root.pem')

sslsock = context.wrap_socket(sock, server_hostname=HOST)
...
```

Servidor

```
import ssl
...
context = ssl.SSLContext(protocol=ssl.PROTOCOL_TLS_SERVER)
context.verify_mode = ssl.CERT_NONE
context.load_cert_chain(certfile='serv.crt', keyfile='serv.key')
sslsock = context.wrap_socket(conn_sock, server_side=True)
...
```

Note que o cliente deve ter acesso ao certificado da CA para verificar o certificado que o servidor envia (uma vez que este é assinado pela CA), enquanto o servidor deve ter acesso a sua chave privada (para provar a sua identidade) e ao seu certificado (para ser enviado aos clientes).

3.2. Autenticação Mútua (Servidor e Cliente autenticam-se um para o outro)

Os próximos dois trechos de código apresentam um código similar aos anteriores, mas agora para autenticação mútua entre cliente e servidor.

Cliente

```
import ssl
...
context = ssl.SSLContext(protocol=ssl.PROTOCOL_TLS_CLIENT)
context.verify_mode = ssl.CERT_REQUIRED
context.check_hostname = True
context.load_verify_locations(cafile='root.pem')
context.load_cert_chain(certfile='cli.crt', keyfile='cli.key')

sslsock = context.wrap_socket(sock, server_hostname=HOST)
...
```

Servidor

```
import ssl
...
context = ssl.SSLContext(protocol=ssl.PROTOCOL_TLS_SERVER)
context.verify_mode = ssl.CERT_REQUIRED
context.load_verify_locations(cafile='root.pem')
context.load_cert_chain(certfile='serv.crt', keyfile='serv.key')

sslsock = context.wrap_socket(conn_sock, server_side=True)
...
```

Nestes trechos de código os dois lados requerem três informações: `cafile` (certificado do CA), `certfile` (o certificado a ser enviado no protocolo) e `keyfile` (chave privada usada para provar a identidade).

4. Exemplo de aplicação Flask com autenticação mútua

As listagens a seguir apresentam exemplos de uma aplicação servidor Flask e um cliente com autenticação mútua.

Servidor

```
from flask import Flask
import ssl

app = Flask(__name__)

@app.route('/login', methods = ["GET"])
def login():
    return 'Login'

if __name__ == '__main__':
    context = ssl.SSLContext(protocol=ssl.PROTOCOL_TLS_SERVER)
    context.verify_mode = ssl.CERT_REQUIRED
    context.load_verify_locations(cafile='root.pem')
    context.load_cert_chain(certfile='serv.crt',keyfile='serv.key')
    app.run('localhost', ssl_context=context, debug = True)
```

Cliente

```
import requests

r=requests.get('https://localhost:5000/login',verify='root.pem',cert=('cli.crt','cli.key'))
print (r.status_code)
print (r.content.decode())
print (r.headers)
print ('***')
```

5. Criação de uma chave para o utilizador fazer o login no navegador

5.1 Para o utilizador autenticar-se para o servidor e fazer o login através do navegador, é preciso que o mesmo crie um ficheiro PKCS 12 (extensão `.p12`) que permite acrescentar tanto a chave quanto o certificado do cliente. Para criar este ficheiro basta executar o seguinte comando (não precisa colocar a senha, pois a chave `cli.key` foi criada sem esta):

```
$ openssl pkcs12 -export -inkey cli.key -in cli.crt -out cli.p12
```

5.2 A seguir, é preciso importar este ficheiro no navegador. Para fazê-lo por exemplo no Firefox:
a) Abrir as preferências do navegador ou uma aba com o endereço `about:preferences`.
b) Aceder ao item "`Privacy & Security`".
c) Na parte dos Certificados, carregar em "`View Certificates`".
d) Selecionar a aba "`Your Certificates`" e carregar no botão "`Import`".
e) Selecionar o ficheiro `cli.p12` criado no passo 3.1. Novamente não precisa inserir senha. Carregar em `Ok` para fechar a janela.
f) Quando o servidor Flask estiver implementado e a ser executado, o cliente pode aceder à página HTTPS para fazer o login: https://localhost:5000/login.
g) Neste caso, quando o site pedir ao utilizador para autenticar-se para o servidor, carregar em `Ok`.

---

6. Exercícios

1. Siga os passos da Secção 2 para criar todas as chaves e certificados necessários para utilizar comunicação segura entre servidores e clientes em Python.

2. Copie os programas cliente e servidor apresentados a seguir e execute-os no computador. Note que este programa consiste na versão final do programa construído nos guiões anteriores (e depois usado em guiões posteriores). Pode-se utilizar o Wireshark para monitorizar a comunicação entre o cliente e o servidor.

Cliente

```
import sys, socket as s

if len(sys.argv) > 1:
    HOST = sys.argv[1]
    PORT = int(sys.argv[2])
else:
    HOST = 'localhost'
    PORT = 9999

while True:
    msg = input('Mensagem: ');

    if msg == 'EXIT':
        exit()

    sock = s.socket(s.AF_INET, s.SOCK_STREAM)
    sock.connect((HOST, PORT))

    sock.sendall(msg.encode())
    resposta = sock.recv(1024)

    print ('Recebi: %s' % resposta.decode())
    sock.close()
```

Servidor

```
import sys, socket as s

HOST = 'localhost'
if len(sys.argv) > 1:
        PORT = int(sys.argv[1])
else:
        PORT = 9999
sock = s.socket(s.AF_INET, s.SOCK_STREAM)
sock.setsockopt(s.SOL_SOCKET, s.SO_REUSEADDR, 1)

sock.bind((HOST, PORT))
sock.listen(1)

list = []

while True:
        (conn_sock, addr) = sock.accept()
        try:
                tmp = conn_sock.recv(1024)
                msg = tmp.decode()
                resp = 'Ack'

                if msg == 'LIST':
                        resp = str(list)
                elif msg == 'CLEAR':
                        list = []
                        resp = "Lista apagada"
                else:
                        list.append(msg)

                conn_sock.sendall(resp.encode())

                print ('list= %s' % list)
                conn_sock.close()
        except:
                print ('socket fechado!')
                conn_sock.close()

sock.close()
```

3. Modifique os programas anteriores para usarem SSL para comunicação, de tal forma que o cliente verifique a identidade do servidor (autenticação unilateral).

4. Modifique os programas anteriores para usarem SSL para comunicação com autenticação mútua. Novamente, pode-se utilizar o Wireshark para confirmar que o protocolo de inicialização do TLS foi executado e que a comunicação agora ocorre de maneira segura.

5. Copie os programas da Secção 4 e execute-os. Para verificar que o servidor exige autenticação mútua, pode-se remover o campo `cert` do pedido feito pelo cliente.

6. Siga os passos da Secção 5 para que um utilizador consiga fazer pedidos ao servidor através do navegador.

---

7. Bibliografia e outro material de apoio

- https://www.openssl.org/
- https://www.openssl.org/docs/apps/openssl.html
- https://docs.python.org/3/library/ssl.html