TLS1.3实验报告

张志豪 20307130085

1. 实验目的

简单模拟地实现 TLS 的流程和最新的 TLS 1.3，使得：

1、客户端和服务器可以通过socket建立连接实现相互通信

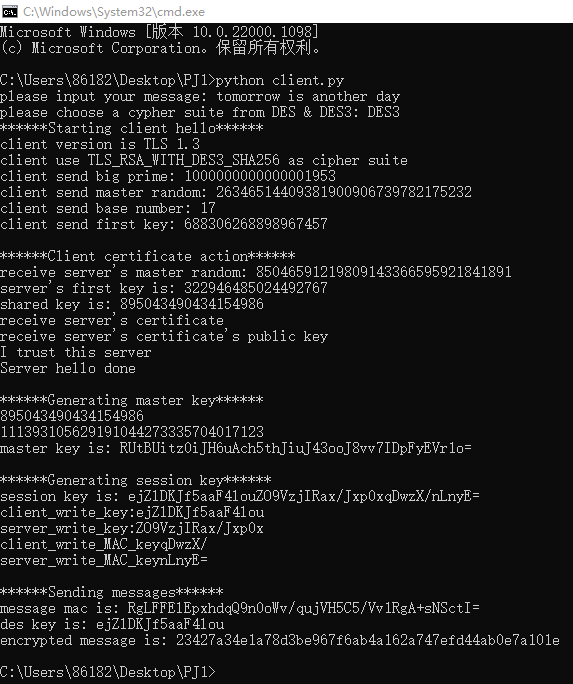
2、客户端和服务器应就加密的会话密钥（session key）达成一致，会话密钥应显示在服务器的屏幕或用户界面上。

3、客户端加密一个字符串，计算 MAC 并发送给服务器。

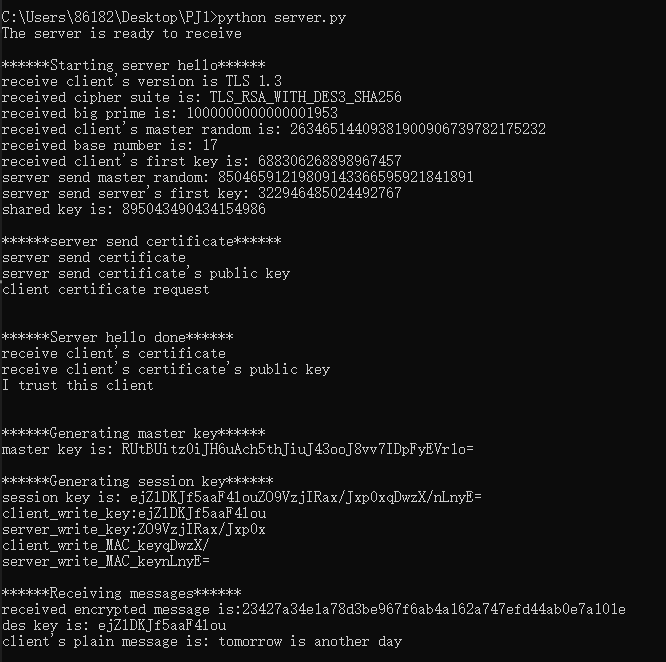
4、服务器解密，验证 MAC。

1. 实验结果

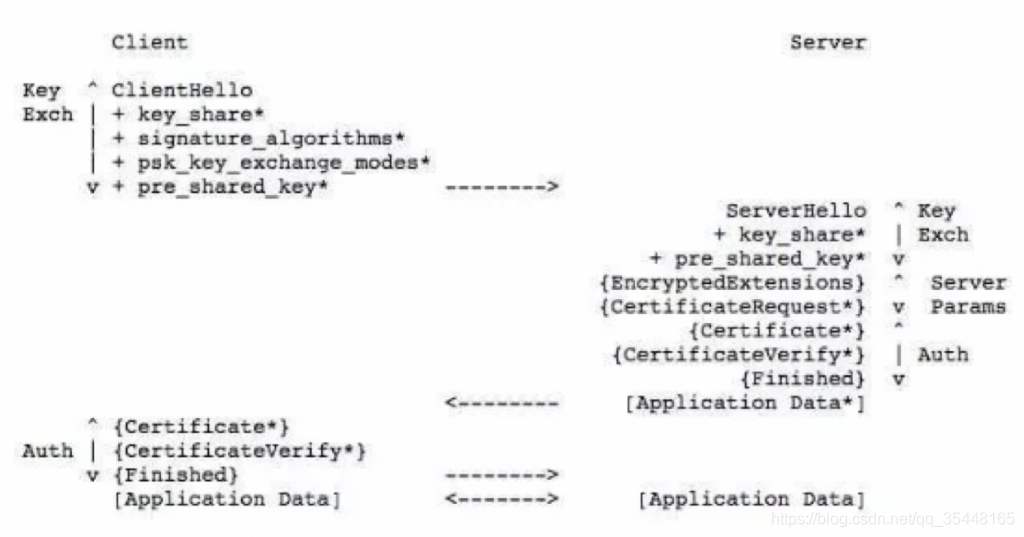
Client端：



Server端：



1. 实验步骤



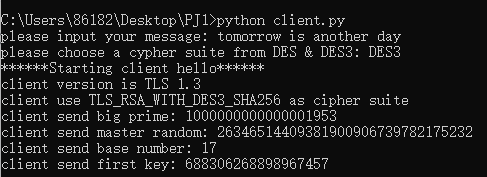
与TLS1.2相比，TLS1.3的密钥协商过程只需要一个RTT，通过DH算法生成pre-master key，然后在client端和server端各自生成master key和session key。

1. client hello

当客户端第一次连接到服务器时，它需要发送ClientHello作为它的第一条消息。在client hello中需要发送版本号、选择的加密套件（用于校验）；同时发送client端选择的大质数P，底数G和运算A=G^random%P之后的结果（用于进行DH算法）；还要发送一个大随机数（用于生成master key）

1. def ClientHello() -> None:
2. print("\*\*\*\*\*\*Starting client hello\*\*\*\*\*\*")
3. # exchange version
4. client\_version = "TLS 1.3"
5. print("client version is " + client\_version)
6. clientSocket.send(client\_version.encode('UTF-8'))
7. time.sleep(1)
8. # exchange cipher suite
9. global cipher\_suite, used\_cipher\_suite
10. **if** cipher\_suite == "DES":
11. used\_cipher\_suite = "TLS\_RSA\_WITH\_DES\_SHA256"
12. elif cipher\_suite == "DES3":
13. used\_cipher\_suite = "TLS\_RSA\_WITH\_DES3\_SHA256"
14. print("client use " + used\_cipher\_suite + " as cipher suite")
15. clientSocket.send(used\_cipher\_suite.encode('UTF-8'))
16. time.sleep(1)
17. # calculate first key
18. global client\_random
19. global client\_master\_random
20. client\_random = random.randint(100, 999)
21. client\_first\_key = (base \*\* client\_random) % BigPrime
22. # send big prime
23. print("client send big prime: " + str(BigPrime))
24. clientSocket.send(str(BigPrime).encode('UTF-8'))
25. time.sleep(1)
26. # send a big random to sever(used in later progress)
27. client\_master\_random = random.randint(10000000000, 99999999999) \*\* 3
28. print("client send master random: " + str(client\_master\_random))
29. clientSocket.send(str(client\_master\_random).encode('UTF-8'))
30. time.sleep(1)
31. # send base number
32. print("client send base number: " + str(base))
33. clientSocket.send(str(base).encode('UTF-8'))
34. time.sleep(1)
35. # send client's first key
36. print("client send first key: " + str(client\_first\_key) + "\n")
37. clientSocket.send(str(client\_first\_key).encode('UTF-8'))
38. time.sleep(1)

运行结果如图：

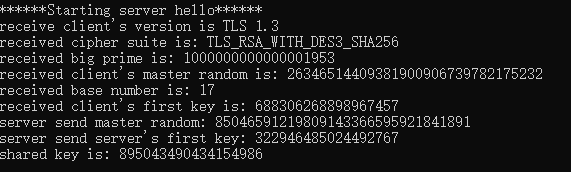


1. server hello

当server接收到client发过来的各种信息时要完成一下操作：（1）验证版本号是不是TLS1.3；（2）加密套件是不是DES和DES3中的一种；（3）利用client传过来的大质数、底数和运算结果A，结合自己生成的随机数根据DH算法产生pre-master key。

1. def ServerHello() -> None:
2. print("\*\*\*\*\*\*Starting server hello\*\*\*\*\*\*")
3. # receive client version
4. received\_client\_version = connectionSocket.recv(1024).decode('UTF-8')
5. print("receive client's version is " + received\_client\_version)
6. # receive client's cipher suite
7. received\_cipher\_suite = connectionSocket.recv(1024).decode('UTF-8')
8. print("received cipher suite is: " + received\_cipher\_suite)
9. time.sleep(1)
10. global cipher\_suite
11. **if** received\_cipher\_suite == "TLS\_RSA\_WITH\_DES\_SHA256":
12. cipher\_suite = "DES"
13. elif received\_cipher\_suite == "TLS\_RSA\_WITH\_DES3\_SHA256":
14. cipher\_suite = "DES3"
15. **else**:
16. print("cipher suite wrong")
17. serverSocket.close()
18. # receive big prime
19. big\_prime = **int**(connectionSocket.recv(1024).decode('UTF-8'))
20. print("received big prime is: " + str(big\_prime))
21. global received\_client\_master\_random
22. # receive client's master random(used in later progress)
23. received\_client\_master\_random = **int**(
24. connectionSocket.recv(1024).decode('UTF-8'))
25. print("received client's master random is: " +
26. str(received\_client\_master\_random))
27. # receive base number
28. base\_num = **int**(connectionSocket.recv(1024).decode('UTF-8'))
29. print("received base number is: " + str(base\_num))
30. # receive client's first key
31. client\_first\_key = **int**(connectionSocket.recv(1024).decode('UTF-8'))
32. print("received client's first key is: " + str(client\_first\_key))
33. global server\_master\_random
34. # generate server's random number for DH and send it
35. server\_master\_random = random.randint(10000000000, 99999999999) \*\* 3
36. print("server send master random: " + str(server\_master\_random))
37. connectionSocket.send(str(server\_master\_random).encode('UTF-8'))
38. time.sleep(1)
39. server\_random = random.randint(100, 999)
40. # calculate pre-master key
41. global premaster\_key
42. premaster\_key = (client\_first\_key \*\* server\_random) % big\_prime
43. server\_first\_key = (base\_num \*\* server\_random) % big\_prime
44. print("server send server's first key: " + str(server\_first\_key))
45. connectionSocket.send(str(server\_first\_key).encode('UTF-8'))
46. print("shared key is: " + str(premaster\_key) + "\n")
47. time.sleep(1)

运行结果如下图：

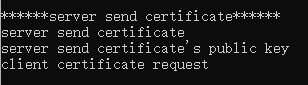


1. server cert

接下来还是在server端进行，将自己的证书和公钥发送给client进行验证，并请求client端的证书。

1. def ServerCert() -> None:  # send server's certificate
2. print("\*\*\*\*\*\*server send certificate\*\*\*\*\*\*")
3. # server\_cert\_path = 'key/server/server.crt'
4. server\_cert\_public\_key\_path = 'key/server/server\_public.key'
5. print("server send certificate")
6. # connectionSocket.send(server\_cert\_path.encode('UTF-8'))
7. # time.sleep(1)
8. print("server send certificate's public key")
9. connectionSocket.send(server\_cert\_public\_key\_path.encode('UTF-8'))
10. time.sleep(1)
11. print("client certificate request")
12. cer\_req = "please send me your certificate"
13. print('\n')
14. connectionSocket.send(cer\_req.encode('UTF-8'))

运行结果如下图：

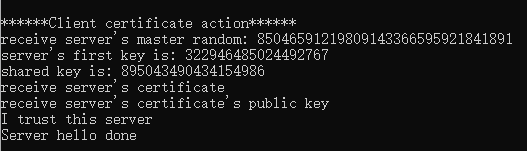


1. client cert

Client首先接收来自server的DH运算中间值，利用该值和自己的P、G、random运算生成pre-master key；然后Client接收server发过来的证书和公钥，从证书中提取公钥，与发送过来的公钥作比较，如果相同，则trust server；在接收到server端的证书请求时，向server发送自己的证书及公钥。

1. def ClientCert() -> None:
2. print("\*\*\*\*\*\*Client certificate action\*\*\*\*\*\*")
3. global received\_server\_master\_random
4. received\_server\_master\_random = **int**(clientSocket.recvfrom(1024)[0].decode('UTF-8'))
5. print("receive server's master random: " + str(received\_server\_master\_random))
6. received\_server\_first\_key = **int**(clientSocket.recvfrom(1024)[0].decode('UTF-8'))
7. print("server's first key is: " + str(received\_server\_first\_key))
8. global premaster\_key
9. premaster\_key = (received\_server\_first\_key \*\* client\_random) % BigPrime
10. print("shared key is: " + str(premaster\_key))
11. # received\_server\_cert = clientSocket.recvfrom(1024)[0].decode('UTF-8')
12. received\_server\_cert = 'key/server/server.crt'
13. print("receive server's certificate")
14. received\_server\_cert\_public\_key\_path = clientSocket.recvfrom(1024)[0].decode('UTF-8')
15. server\_cert\_public\_key = open(received\_server\_cert\_public\_key\_path).read()
16. print("receive server's certificate's public key")
17. server\_public\_key = read\_public\_key\_from\_crt(received\_server\_cert)
18. **if** server\_public\_key.decode() == server\_cert\_public\_key:
19. print("I trust this server")
20. **else**:
21. print("I don't trust this server")
22. clientSocket.close()
23. received\_cert\_request = clientSocket.recvfrom(1024)[0].decode('UTF-8')
24. **if** received\_cert\_request == "please send me your certificate":
25. client\_cert\_path = 'key/client/client.crt'
26. client\_cert\_public\_key\_path = 'key/client/client\_public.key'
27. clientSocket.send(client\_cert\_path.encode('UTF-8'))
28. time.sleep(1)
29. clientSocket.send(client\_cert\_public\_key\_path.encode('UTF-8'))
30. time.sleep(1)
31. received\_server\_hello\_done = clientSocket.recvfrom(1024)[0].decode('UTF-8')
32. print(received\_server\_hello\_done + "\n")
33. **else**:
34. print("no certificate request")
35. clientSocket.close()

运行结果如下图：

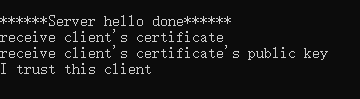


1. server hello done

接收client的证书和公钥，用与之前相同的方法提取公钥并验证。

1. def ServerHelloDone() -> None:
2. print("\*\*\*\*\*\*Server hello done\*\*\*\*\*\*")
3. print("receive client's certificate")
4. received\_client\_cert\_path = connectionSocket.recv(1024).decode('UTF-8')
5. print("receive client's certificate's public key")
6. received\_client\_cert\_public\_key\_path = connectionSocket.recv(
7. 1024).decode('UTF-8')
8. client\_cert\_public\_key = read\_public\_key\_from\_crt(
9. received\_client\_cert\_path)
10. client\_public\_key = open(received\_client\_cert\_public\_key\_path).read()
11. **if** client\_cert\_public\_key.decode() == client\_public\_key:
12. print("I trust this client")
13. **else**:
14. print("I don't trust this client")
15. serverSocket.close()
16. connectionSocket.send("Server hello done".encode('UTF-8'))
17. print("\n")

运行结果如下图：

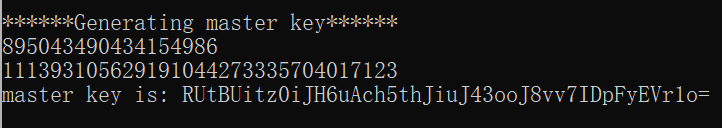


1. Generate master key（client和server一样）

利用之前已经生成的pre-master key，以及第一步在ClientHello和ServerHello传递的大随机数，在client和server端各自产生master key。过程中利用的SHA256算法调用了库函数hmac。

1. def GenerateMasterKey() -> None:
2. print("\*\*\*\*\*\*Generating master key\*\*\*\*\*\*")
3. global master\_key
4. master\_key = get\_sha256(str(premaster\_key), str(
5. received\_client\_master\_random + server\_master\_random))
6. print("master key is: " + master\_key + "\n")
7. def get\_sha256(data, key) -> string:    # SHA256 encryption
8. key = key.encode('utf-8')  # sha256加密的key
9. message = data.encode('utf-8')  # 待sha256加密的内容
10. sign = base64.b64encode(
11. hmac.**new**(key, message, digestmod=sha256).digest()).decode()
12. **return** sign

运行结果如图所示：

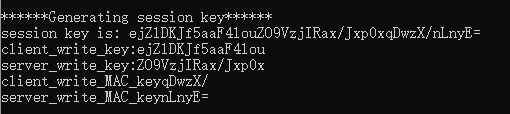


1. generate session key（client端和server端一样）

利用上一步中生成的master key产生session key；其中的加密套件使用DES还是DES3由ClientHello中选择。其中DES和DES3的区别在于用来加密的字符串长短不同，DES为8位，而DES3为16位。

1. def GenerateSessionKey() -> None:
2. global session\_key
3. session\_key = get\_sha256(master\_key, str(
4. received\_client\_master\_random + server\_master\_random))
5. print("session key is: " + session\_key)
6. global client\_write\_key, server\_write\_key, client\_write\_MAC\_key, server\_write\_MAC\_key
7. **if** cipher\_suite == "DES":
8. client\_write\_key = session\_key[:8]
9. print("client\_write\_key:" + str(client\_write\_key))
10. server\_write\_key = session\_key[8:16]
11. print("server\_write\_key:" + str(server\_write\_key))
12. client\_write\_MAC\_key = session\_key[16:29]
13. print("client\_write\_MAC\_key" + str(client\_write\_MAC\_key))
14. server\_write\_MAC\_key = session\_key[29:]
15. print("server\_write\_MAC\_key" + str(server\_write\_MAC\_key) + "\n")
16. elif cipher\_suite == "DES3":
17. client\_write\_key = session\_key[:16]
18. print("client\_write\_key:" + str(client\_write\_key))
19. server\_write\_key = session\_key[16:32]
20. print("server\_write\_key:" + str(server\_write\_key))
21. client\_write\_MAC\_key = session\_key[32:38]
22. print("client\_write\_MAC\_key" + str(client\_write\_MAC\_key))
23. server\_write\_MAC\_key = session\_key[38:]
24. print("server\_write\_MAC\_key" + str(server\_write\_MAC\_key) + "\n")

运行结果如图所示：

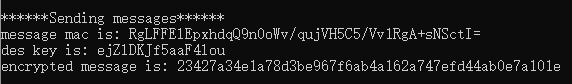


1. Send message

用前一步生成的session key截取一部分定义为client\_write\_MAC\_key（DES和DES3截取的长度就不一样了，因为DES3用于des加密的长度更多），通过SHA256生成MAC；再用des算法加密用户输入内容，并发送给server。

1. def SendMessages() -> None:
2. print("\*\*\*\*\*\*Generating session key\*\*\*\*\*\*")
3. GenerateSessionKey()
4. print("\*\*\*\*\*\*Sending messages\*\*\*\*\*\*")
5. global client\_write\_key, server\_write\_key, client\_write\_MAC\_key, server\_write\_MAC\_key
6. global message
7. message\_mac = base64.b64encode(hmac.**new**(client\_write\_MAC\_key.encode(), message.encode(), digestmod=sha256).digest())
8. print("message mac is: " + message\_mac.decode())
9. des\_key = client\_write\_key
10. print("des key is: " + des\_key)
11. encrypted\_message = des\_en(des\_key, message, cipher\_suite)
12. print("encrypted message is: " + encrypted\_message)
13. clientSocket.send(encrypted\_message.encode())
14. time.sleep(1)

运行结果如图所示：

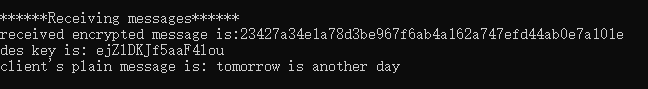


1. receive message

Server端同样利用之前生成的session key截取片段用来解密，最后输出client端用户输入的plain message。

1. def ReceiveMessage() -> None:
2. print("\*\*\*\*\*\*Generating session key\*\*\*\*\*\*")
3. time.sleep(1)
4. GenerateSessionKey()
5. global client\_write\_key, server\_write\_key, client\_write\_MAC\_key, server\_write\_MAC\_key
6. encrypted\_message = connectionSocket.recv(1024)
7. print("\*\*\*\*\*\*Receiving messages\*\*\*\*\*\*")
8. print("received encrypted message is:" + encrypted\_message.decode())
9. des\_key = client\_write\_key
10. print("des key is: " + des\_key)
11. decrypted\_message = des\_de(des\_key, encrypted\_message, cipher\_suite)
12. print("client's plain message is: " + decrypted\_message)

运行结果如图所示，输出了用户输入的“tomorrow is another day”：



1. 实验思考
2. TLS1.3和TLS1.2相比，在生成pre-master key的过程中利用DH算法是更安全高效的（TLS1.2使用的是RSA，根据RSA算法的特性，如果Server的私钥泄露了，那么以前劫持的会话记录都可以被解析），并且节省了一个RTT，但是后续生成master密钥的过程可能较复杂（在本project中后续密钥生成使用的是TLS1.2的方法）
3. TLS虽然是比较成熟的协议，但是依然无法阻止中间人攻击。