山东大学 软件 学院

**Python高级程序设计(网络空间安全)**课程实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：202200201095 | 姓名：杨伟康 | | 班级： 网安班 |
| 实验题目：实验6-8公钥和对称密码结合，认证加密方案，TLS通信 | | | |
| 实验学时：2 | | 实验日期： 2024.11.6 | |
| 实验目的：   * 综合理解公钥密码和对称密码的结合   + 理解公钥密码在通信中加密实现会话密钥分发、数字签名实现消息认证的作用。   + 理解对称密码在通信中加密实现会话信息保密、MAC实现会话信息认证的作用。 * 理解DH密钥交换协议的对称密钥安全分发功能及前向安全性。 * 理解对称加密体制的认证加密模式的功能及实现原理。 * 理解SSL/TLS建立安全通信的原理。 | | | |
| 硬件环境：  PC机 | | | |
| 软件环境：  **Pycharm** + Python | | | |
| 实验步骤与内容： 学习并测试代码7-4、7-5服务器和客户端网络通信代码，实现以下2、3中的网络通信利用RSA公钥密码和AES和HMAC算法实现从对称密钥分发到安全会话（实现两方的通信）。 实现上面通信需要几个基本步骤：   1. 生成RSA密钥对（公钥和私钥）。   私钥服务端自己储存，公钥以文件形式存在文件夹中，相当于将公钥分发出去，       1. 客户端使用服务器的公钥加密对称密钥（AES密钥）。      1. 客户端将加密后的对称密钥和其他数据（如nonce、密文和HMAC标签）一起发送给服务器。      1. 服务器使用其私钥解密接收到的对称密钥，然后使用解密后的密钥解密消息并验证HMAC。       客户端和服务器通信代码运行实例   1. 开启服务端      1. 客户端发送消息      1. 服务端收到      1. 客户端得到回复     服务端代码： import asyncio import hashlib import hmac import os import sys from cryptography.hazmat.primitives.ciphers.aead import AESGCM from cryptography.hazmat.primitives import hashes from cryptography.hazmat.primitives.kdf.hkdf import HKDF from cryptography.hazmat.backends import default\_backend from cryptography.hazmat.primitives.asymmetric import rsa, padding from cryptography.hazmat.primitives import serialization  PW = b"password"  # Generate RSA key pair (for demonstration purposes only) def generate\_rsa\_key\_pair():  private\_key = rsa.generate\_private\_key(  public\_exponent=65537,  key\_size=2048,  backend=default\_backend()  )  public\_key = private\_key.public\_key()  return private\_key, public\_key  private\_key, public\_key = generate\_rsa\_key\_pair()  # Save public key to a file (or use another method to share it with clients) with open("server\_public\_key.pem", "wb") as f:  f.write(public\_key.public\_bytes(  encoding=serialization.Encoding.PEM,  format=serialization.PublicFormat.SubjectPublicKeyInfo  ))  class EchoServerProtocol(asyncio.Protocol):  def \_\_init\_\_(self):  self.private\_key = private\_key   def connection\_made(self, transport):  peername = transport.get\_extra\_info('peername')  print('Connection from {}'.format(peername))  self.transport = transport   def data\_received(self, data):  # Extract encrypted key, nonce, ciphertext, and HMAC tag  encrypted\_key\_length = 256 # Length of the RSA-encrypted AES key (2048 bits / 8 bytes = 256 bytes)  encrypted\_key = data[:encrypted\_key\_length]  data = data[encrypted\_key\_length:]  nonce, ciphertext, hmac\_tag = data[:12], data[12:-32], data[-32:]  aad = b"" # Additional Authenticated Data, not used in this example   # Decrypt the AES key using RSA private key  aes\_key = self.private\_key.decrypt(  encrypted\_key,  padding.OAEP(  mgf=padding.MGF1(algorithm=hashes.SHA256()),  algorithm=hashes.SHA256(),  label=None  )  )   # Verify HMAC  received\_hmac = hmac\_tag  expected\_hmac = hmac.new(aes\_key, ciphertext, hashlib.sha256).digest()  if not hmac.compare\_digest(received\_hmac, expected\_hmac):  print("HMAC verification failed!")  self.transport.close()  return   # Decrypt the ciphertext  plaintext = AESGCM(aes\_key).decrypt(nonce, ciphertext, aad)  message = plaintext.decode()  print('Decrypted message from client: {!r}'.format(message))   # Echo back message  print('Echo back message: {!r}'.format(message))  reply\_nonce = os.urandom(12)  reply\_ciphertext = AESGCM(aes\_key).encrypt(reply\_nonce, message.encode(), aad)  reply\_hmac = hmac.new(aes\_key, reply\_ciphertext, hashlib.sha256).digest()  self.transport.write(reply\_nonce + reply\_ciphertext + reply\_hmac)   # Close the client socket  self.transport.close()  # FOR AUTO TESTING. Shutdown after echo  if "--auto-test" in sys.argv:  print("[PASS]")  asyncio.get\_event\_loop().call\_later(0.25, sys.exit)  loop = asyncio.get\_event\_loop() coro = loop.create\_server(lambda: EchoServerProtocol(), '127.0.0.1', 8888) server = loop.run\_until\_complete(coro)  print('Serving on {}'.format(server.sockets[0].getsockname())) try:  loop.run\_forever() except KeyboardInterrupt:  pass  server.close() loop.run\_until\_complete(server.wait\_closed()) loop.close()  客户端代码：  import asyncio import hashlib import hmac import os import sys from cryptography.hazmat.primitives.ciphers.aead import AESGCM from cryptography.hazmat.primitives import hashes from cryptography.hazmat.primitives.kdf.hkdf import HKDF from cryptography.hazmat.backends import default\_backend from cryptography.hazmat.primitives.asymmetric import rsa, padding from cryptography.hazmat.primitives import serialization  PW = b"password"  # Load server's public key from a file with open("server\_public\_key.pem", "rb") as f:  server\_public\_key = serialization.load\_pem\_public\_key(  f.read(),  backend=default\_backend()  )  class EchoClientProtocol(asyncio.Protocol):  def \_\_init\_\_(self, message):  self.message = message   # Derive AES key using HKDF  key\_material = HKDF(  algorithm=hashes.SHA256(),  length=32, salt=None, info=None,  backend=default\_backend()  ).derive(PW)  self.\_aes\_key = key\_material   def connection\_made(self, transport):  plaintext = self.message.encode()  nonce = os.urandom(12)  aad = b"" # Additional Authenticated Data, not used in this example   # Encrypt AES key using server's public key  encrypted\_aes\_key = server\_public\_key.encrypt(  self.\_aes\_key,  padding.OAEP(  mgf=padding.MGF1(algorithm=hashes.SHA256()),  algorithm=hashes.SHA256(),  label=None  )  )   # Encrypt the message using AES-GCM  ciphertext = AESGCM(self.\_aes\_key).encrypt(nonce, plaintext, aad)   # Calculate HMAC  hmac\_tag = hmac.new(self.\_aes\_key, ciphertext, hashlib.sha256).digest()   # Send encrypted AES key, nonce, ciphertext, and HMAC tag  transport.write(encrypted\_aes\_key + nonce + ciphertext + hmac\_tag)  print('Encrypted data sent: {!r}'.format(self.message))   def data\_received(self, data):  nonce, ciphertext, hmac\_tag = data[:12], data[12:-32], data[-32:]  aad = b"" # Additional Authenticated Data, not used in this example   # Verify HMAC  received\_hmac = hmac\_tag  expected\_hmac = hmac.new(self.\_aes\_key, ciphertext, hashlib.sha256).digest()  if not hmac.compare\_digest(received\_hmac, expected\_hmac):  print("HMAC verification failed!")  return   # Decrypt the ciphertext  plaintext = AESGCM(self.\_aes\_key).decrypt(nonce, ciphertext, aad)  print('Decrypted response from server: {!r}'.format(plaintext.decode()))  if "--auto-test" in sys.argv:  if plaintext.decode() == self.message:  print("[PASS]")  else:  print("[FAIL]")  loop = asyncio.get\_event\_loop() message = sys.argv[1] coro = loop.create\_connection(lambda: EchoClientProtocol(message), '127.0.0.1', 8888) loop.run\_until\_complete(coro) loop.run\_forever() loop.close() 编程实现抵御中间人攻击的ECDH密钥交换（实现两方的通信）  这次收发的消息内容我直接放入了代码当中  并且使服务器发送的和客户端发送的代码不同，检验是否能够正确加密和解密不同消息  ECDH密钥交换分为下面几步   1. 服务端和客户端生成自己的公钥和私钥        1. 将各自的公钥发送给对方      1. 生成共享密钥，这里把最终生成的密钥打印出来，检测双方的共享密钥是否一致      1. 最后发送信息   启动服务端    启动客户端，服务端接收并且回复消息      服务端：  from cryptography.hazmat.primitives.asymmetric import ec from cryptography.hazmat.primitives import serialization from cryptography.hazmat.primitives.kdf.hkdf import HKDF from cryptography.hazmat.primitives import hashes from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes from cryptography.hazmat.backends import default\_backend import socket import os  # 生成服务器端的私钥和公钥 private\_key = ec.generate\_private\_key(ec.SECP384R1(), default\_backend()) public\_key = private\_key.public\_key()  # 将公钥序列化为PEM格式，以便发送 pem\_public\_key = public\_key.public\_bytes(  encoding=serialization.Encoding.PEM,  format=serialization.PublicFormat.SubjectPublicKeyInfo )  # 创建socket server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) server\_socket.bind(('localhost', 8808)) server\_socket.listen(1) print("Server listening on port 8808")  # 接受客户端连接 client\_socket, client\_address = server\_socket.accept() print(f"Connection from {client\_address}")  # 接收客户端的公钥 pem\_data = client\_socket.recv(1024).decode('utf-8') client\_public\_key = serialization.load\_pem\_public\_key(  pem\_data.encode('utf-8'),  backend=default\_backend() )  # 发送公钥到客户端（直接发送二进制数据） client\_socket.sendall(pem\_public\_key)  # 生成共享密钥 shared\_key = private\_key.exchange(ec.ECDH(), client\_public\_key)  # 使用HKDF派生出一个密钥 derived\_key = HKDF(  algorithm=hashes.SHA256(),  length=32,  salt=None,  info=b'handshake data',  backend=default\_backend() ).derive(shared\_key)  print(f"Server derived key: {derived\_key.hex()}")  # 准备AES加密的iv（初始化向量） iv = os.urandom(16) # AES的块大小是16字节  # 加密消息 cipher = Cipher(algorithms.AES(derived\_key), modes.CFB(iv), backend=default\_backend()) encryptor = cipher.encryptor() message = b"Hello, this is a secret message!" ciphertext = encryptor.update(message) + encryptor.finalize()  # 发送iv和密文给客户端 client\_socket.sendall(iv + ciphertext)  # 接收客户端发送回来的相同消息（已加密） received\_iv\_and\_ciphertext = client\_socket.recv(1024) received\_iv = received\_iv\_and\_ciphertext[:16] received\_ciphertext = received\_iv\_and\_ciphertext[16:]  # 解密消息 decipher = Cipher(algorithms.AES(derived\_key), modes.CFB(received\_iv), backend=default\_backend()) decryptor = decipher.decryptor() plaintext = decryptor.update(received\_ciphertext) + decryptor.finalize()  print(f"Server received message: {plaintext.decode('utf-8')}")  # 关闭连接 client\_socket.close() server\_socket.close()  客户端from cryptography.hazmat.primitives.asymmetric import ec from cryptography.hazmat.primitives import serialization from cryptography.hazmat.primitives.kdf.hkdf import HKDF from cryptography.hazmat.primitives import hashes from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes from cryptography.hazmat.backends import default\_backend import socket import os  # 生成客户端的私钥和公钥 private\_key = ec.generate\_private\_key(ec.SECP384R1(), default\_backend()) public\_key = private\_key.public\_key()  # 将公钥序列化为PEM格式，以便发送 pem\_public\_key = public\_key.public\_bytes(  encoding=serialization.Encoding.PEM,  format=serialization.PublicFormat.SubjectPublicKeyInfo )  # 创建socket并连接到服务器 client\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) client\_socket.connect(('localhost', 8808))  # 发送公钥到服务器（直接发送二进制数据） client\_socket.sendall(pem\_public\_key)  # 接收服务器的公钥（直接接收二进制数据） pem\_data = client\_socket.recv(1024) server\_public\_key = serialization.load\_pem\_public\_key(  pem\_data,  backend=default\_backend() )  # 生成共享密钥 shared\_key = private\_key.exchange(ec.ECDH(), server\_public\_key)  # 使用HKDF派生出一个密钥 derived\_key = HKDF(  algorithm=hashes.SHA256(),  length=32,  salt=None,  info=b'handshake data',  backend=default\_backend() ).derive(shared\_key)  print(f"Client derived key: {derived\_key.hex()}")  # 接收服务器发送的iv和密文 received\_iv\_and\_ciphertext = client\_socket.recv(1024) received\_iv = received\_iv\_and\_ciphertext[:16] received\_ciphertext = received\_iv\_and\_ciphertext[16:]  # 解密消息 decipher = Cipher(algorithms.AES(derived\_key), modes.CFB(received\_iv), backend=default\_backend()) decryptor = decipher.decryptor() plaintext = decryptor.update(received\_ciphertext) + decryptor.finalize()  print(f"Client received message: {plaintext.decode('utf-8')}")  # 定义要发送的消息 message = "Hello, Server!"  # 准备发送相同的消息给服务器（加密） cipher = Cipher(algorithms.AES(derived\_key), modes.CFB(received\_iv), backend=default\_backend()) encryptor = cipher.encryptor() ciphertext\_to\_send = encryptor.update(message.encode('utf-8')) + encryptor.finalize()  # 发送iv和密文给服务器（注意：这里为了简化，我们重用了服务器的iv。在实际应用中，应该为每个消息生成新的iv） client\_socket.sendall(received\_iv + ciphertext\_to\_send)  # 关闭连接 client\_socket.close()  ： 编写使用AES-GCM模式实现对消息进行“附加数据的身份认证加密” from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes from cryptography.hazmat.primitives import padding from cryptography.hazmat.backends import default\_backend import os  def encrypt\_message(key, nonce, plaintext, aad):  # 创建Cipher对象，使用AES-GCM模式  cipher = Cipher(algorithms.AES(key), modes.GCM(nonce), backend=default\_backend())  encryptor = cipher.encryptor()   # 更新附加数据 (AAD)  encryptor.authenticate\_additional\_data(aad)   # 填充数据到块大小（如果需要）  padder = padding.PKCS7(algorithms.AES.block\_size).padder()  padded\_plaintext = padder.update(plaintext) + padder.finalize()   # 加密数据  ciphertext = encryptor.update(padded\_plaintext) + encryptor.finalize()   # 获取标签  tag = encryptor.tag   return ciphertext, tag  def decrypt\_message(key, nonce, ciphertext, tag, aad):  # 创建Cipher对象，使用AES-GCM模式  cipher = Cipher(algorithms.AES(key), modes.GCM(nonce, tag), backend=default\_backend())  decryptor = cipher.decryptor()   # 更新附加数据 (AAD)  decryptor.authenticate\_additional\_data(aad)   # 解密数据  padded\_plaintext = decryptor.update(ciphertext) + decryptor.finalize()   # 去除填充数据  unpadder = padding.PKCS7(algorithms.AES.block\_size).unpadder()  plaintext = unpadder.update(padded\_plaintext) + unpadder.finalize()   return plaintext  # 示例使用 if \_\_name\_\_ == "\_\_main\_\_":  # 生成一个随机密钥（AES-256需要32字节的密钥）  key = os.urandom(32)   # 生成一个随机nonce（对于AES-GCM，nonce的长度必须是12字节）  nonce = os.urandom(12)   # 示例消息和附加数据  plaintext = b"Hello, this is a secret message!"  aad = b"This is the associated authenticated data."   # 加密消息  ciphertext, tag = encrypt\_message(key, nonce, plaintext, aad)  print(f"Ciphertext: {ciphertext}")  print(f"Tag: {tag}")   # 解密消息  decrypted\_plaintext = decrypt\_message(key, nonce, ciphertext, tag, aad)  print(f"Decrypted Plaintext: {decrypted\_plaintext}")   编程模拟实现SSL通信，即利用证书实现认证和会话密钥协商，进一步实现应用会话密钥进行加密认证安全通信的全过程。 服务端代码：  import socket import ssl  # 证书和私钥路径 CERTFILE = 'server.crt' KEYFILE = 'server.key'   def start\_ssl\_server(host='127.0.0.1', port=8443):  context = ssl.create\_default\_context(ssl.Purpose.CLIENT\_AUTH)  context.load\_cert\_chain(certfile=CERTFILE, keyfile=KEYFILE)   bindsocket = socket.socket()  bindsocket.bind((host, port))  bindsocket.listen(5)   while True:  newsocket, fromaddr = bindsocket.accept()  connstream = context.wrap\_socket(newsocket, server\_side=True)  try:  print(f"Connection from {fromaddr}")  data = connstream.recv(1024)  print(f"Received: {data.decode()}")  connstream.sendall(b"Hello from SSL server")  finally:  connstream.shutdown(socket.SHUT\_RDWR)  connstream.close()   if \_\_name\_\_ == "\_\_main\_\_":  start\_ssl\_server()  客户端代码：  import socket import ssl  # 服务器地址和端口 HOST = '127.0.0.1' PORT = 8443  # 服务器的自签名证书文件路径 SERVER\_CERT\_FILE = 'F:/python密码/exp6/server.crt' # 请替换为实际的文件路径   def start\_ssl\_client():  # 创建一个 SSL 上下文  context = ssl.create\_default\_context(ssl.Purpose.SERVER\_AUTH, cafile=SERVER\_CERT\_FILE)  context.check\_hostname = False # 确保主机名验证是启用的  context.verify\_mode = ssl.CERT\_REQUIRED # 要求服务器提供证书，并且客户端会验证它   # 创建一个套接字并连接到服务器  with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as sock:  sock.connect((HOST, PORT))   # 使用配置好的 SSL 上下文来包装套接字  with context.wrap\_socket(sock, server\_hostname=HOST) as connstream:  try:  print("Sending data...")  connstream.sendall(b"Hello from SSL client")  data = connstream.recv(1024)  print(f"Received: {data.decode()}")  finally:  # 关闭连接  connstream.shutdown(socket.SHUT\_RDWR)   if \_\_name\_\_ == "\_\_main\_\_":  start\_ssl\_client()  使用步骤：  生成自签名证书和私钥  # 生成私钥    # 生成证书签名请求(CSR)，生成自签证书    但是由于这不是正规机构颁发的证书，所以客户端设置有区别  对于测试或开发环境，您可以在客户端上禁用 SSL 证书验证。但是，绝对不要在生产环境中这样做，因为这会使您的连接容易受到中间人攻击。  要在 Python 的 ssl 模块中禁用证书验证，您可以在创建 SSL 上下文时设置 check\_hostname=False 和 verify\_mode=ssl.CERT\_NONE。但是，请再次注意，这样做是不安全的。    后续客户端连接服务器的时候，服务器拒绝我的访问，于是将防火墙关闭    最终成功进行ssl通信，服务端启动    客户端接收消息 | | | |
| 结论分析与体会：  通过本次实验，我成功实现了基于公钥密码和对称密码结合的认证加密方案，并深入理解了TLS通信的原理。具体而言，我达成了以下实验目标：  理解了公钥密码和对称密码的结合：在实验过程中，我利用RSA公钥密码算法实现了会话密钥的安全分发，确保了密钥在传输过程中不被窃听或篡改。同时，我使用AES对称密码算法对会话信息进行加密，保证了通信内容的保密性。  掌握了公钥密码在通信中的作用：通过RSA算法，我实现了会话密钥的加密分发，确保了只有合法的通信双方才能获取密钥。此外，我还了解到数字签名可以用于实现消息认证，确保消息的完整性和来源的真实性。  深入理解了对称密码在通信中的作用：AES算法的高效性和保密性使其成为会话信息加密的理想选择。同时，MAC（消息认证码）的使用进一步增强了会话信息的认证安全性。  掌握了DH密钥交换协议的对称密钥安全分发功能及前向安全性：虽然实验中我使用的是ECDH密钥交换协议，但DH协议的原理与之类似。我了解到ECDH协议能够在不安全的通道上安全地协商出一个共享密钥，且该密钥的协商过程具有前向安全性，即即使未来的私钥被泄露，也无法影响之前通过该密钥加密的通信内容。  理解了对称加密体制的认证加密模式的功能及实现原理：AES-GCM模式将加密和认证结合在一起，不仅实现了对消息的加密，还提供了对消息的完整性认证和来源认证。这种模式的使用大大简化了安全通信的设计和实现。  模拟实现了SSL/TLS通信：通过编程模拟SSL/TLS通信过程，我了解了如何利用证书实现认证和会话密钥协商，并进一步实现了应用会话密钥进行加密认证安全通信的全过程。这一过程让我深刻理解了SSL/TLS协议在保障网络通信安全中的重要作用。  个人体会：  本次实验对我来说是一次宝贵的学习和实践机会。通过亲自编写代码和调试程序，我深入理解了公钥密码和对称密码的结合原理以及TLS通信的实现机制。在实验过程中，我遇到了不少挑战和困难，如密钥分发的不安全、消息认证的失败等。但正是这些挑战促使我不断查阅资料、思考和尝试，最终成功解决了问题。  此外，我还深刻体会到了理论知识与实践操作之间的紧密联系。只有将理论知识应用到实际操作中，才能真正理解和掌握它。同时，我也意识到了自己在编程和算法设计方面的不足，这为我今后的学习和成长指明了方向。  总的来说，这次实验不仅让我掌握了公钥密码和对称密码结合的认证加密方案以及TLS通信的原理，还锻炼了我的编程能力和解决问题的能力。我相信这些经验和知识将对我未来的学习和工作产生积极的影响。 | | | |

注：命名”学号姓名.docx”，例如：“2006999孔明.docx”