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5：利用MAC和数字签名实现消息认证 | | | |
| 实验学时：2 | | 实验日期： 2024.10.30 | |
|  | |  | |
| 实验目的：   * 掌握基于对称密码实现消息认证的MAC设计原理及应用 * 掌握基于非对称密码实现消息认证的数字签名设计原理及应用 * 了解数字证书的结构和作用 | | | |
| 硬件环境：  PC机 | | | |
| 软件环境：  **Pycharm** + Python | | | |
| 实验步骤与内容： 实验内容  1. 利用cryptography中primitives的hashes、hmac模块实现给定消息的认证码生成及验证       运行结果如下，可以看到当我们修改了消息之后HMAC则不匹配，能够有效防止篡改攻击     1. 利用cryptography中primitives的cmac和algrithms.AES实现给定消息的认证码生成及验证       与t1一致，如果message与cmac不一致，则会检测报错   1. 利用cryptography中primitives的asymmetric.rsa生成给定消息的RSA数字签名并完成验证         由于公钥和密钥较长，这里采用pem文件存储  运行结果    私钥内容    公钥内容     1. 利用cryptography中primitives.asymmetric中的ec生成给定消息的椭圆曲线数字签名并完成验证   这里只演示ec的椭圆曲线数字签名及其验证，公钥私钥不再存储     1. 利用Crypto库中的对应功能模块编程实现消息认证码和数字签名的生成和验证   消息认证码MAC    数字签名    运行结果如下     1. 熟悉代码5-6，5-7，5-8，总结数字证书的内容结构、签发方式和作用。   **代码清单5-6 模拟证书颁发 fake\_certs\_issuer.py**  **代码清单5-7 验证模拟证书中的身份 fake\_certs\_prove\_identity.py**  **代码清单5-8 使用证书中公钥对应私钥签名，证明自己的身份； fake\_certs\_verify\_identity.py**  数字证书的内容结构：    数字证书的签发方式：  **生成密钥对**：证书申请者生成一对公私钥，并保留私钥，将公钥提交给证书颁发机构。  **证书请求**：申请者使用私钥对证书请求信息（通常包含公钥和主体信息）进行签名，生成证书签名请求（CSR）。  **审核**：证书颁发机构验证申请者的身份，确保公钥与申请者匹配。  **签发证书**：证书颁发机构使用其私钥对证书内容（包括申请者的公钥、主体信息、颁发者信息、有效期等）进行签名，生成数字证书。  **分发证书**：证书颁发机构将数字证书颁发给申请者，申请者可以将证书分发给需要验证其身份的其他实体。  数字整数的作用：  **数字证书的主要用途**  **将身份和公钥绑定在一起。颁发者可以对证书数据进行签名，以防止修改和提供信任。**  **身份验证**：确保通信双方的身份真实可信，防止中间人攻击。  **数据完整性**：确保传输的数据未被篡改，通过数字签名验证数据的完整性。  **保密性**：使用证书中的公钥加密信息，确保只有持有相应私钥的接收者才能解密。  **防抵赖**：通过数字签名，确保发送者不能否认其发送的信息。  **实验源代码如下：**  from cryptography.hazmat.primitives import hashes, hmac from cryptography.hazmat.backends import default\_backend   def generate\_hmac(key, message):  # 创建一个HMAC对象，指定哈希算法和密钥  hmac\_obj = hmac.HMAC(key, hashes.SHA256(), backend=default\_backend())   # 更新HMAC对象以包含消息内容  hmac\_obj.update(message.encode('utf-8'))   # 获取最终的HMAC值  hmac\_digest = hmac\_obj.finalize()   # 将HMAC值转换为十六进制字符串以便显示和存储  hmac\_hex = hmac\_digest.hex()   return hmac\_hex   def verify\_hmac(key, message, hmac\_hex):  # 将十六进制字符串转换回字节  hmac\_digest = bytes.fromhex(hmac\_hex)   # 创建一个HMAC对象，指定哈希算法和密钥  hmac\_obj = hmac.HMAC(key, hashes.SHA256(), backend=default\_backend())   # 更新HMAC对象以包含消息内容  hmac\_obj.update(message.encode('utf-8'))   # 获取预期的HMAC值  expected\_hmac\_digest = hmac\_obj.finalize()   # 验证HMAC值是否匹配  return hmac\_digest == expected\_hmac\_digest   # 示例用法 key = b'secret\_key' # 密钥（必须是字节类型） message = 'This is a test message.' # 消息  # 生成HMAC hmac\_value = generate\_hmac(key, message) print(f'Generated HMAC: {hmac\_value}')  # 验证HMAC is\_valid = verify\_hmac(key, message, hmac\_value) print(f'HMAC valid: {is\_valid}')  # 修改消息以验证HMAC的完整性 tampered\_message = 'This is a tampered message.' is\_tampered\_valid = verify\_hmac(key, tampered\_message, hmac\_value) print(f'Tampered HMAC valid: {is\_tampered\_valid}')   from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC from cryptography.hazmat.primitives import hashes from cryptography.hazmat.primitives.cmac import CMAC from cryptography.hazmat.backends import default\_backend from base64 import b64encode, b64decode  # 假设你有一个密钥（在实际应用中，应该安全地存储和生成密钥） key = b'Sixteen byte key' # AES-128 需要 16 字节的密钥  # 消息 message = b'This is a secret message'   # 生成CMAC认证码 def generate\_mac(key, message):   cmac = CMAC(algorithms.AES(key), backend=default\_backend())  cmac.update(message)  return cmac.finalize()   # 验证CMAC认证码 def verify\_mac(key, message, mac):  cmac = CMAC(algorithms.AES(key), backend=default\_backend())  cmac.update(message)  return cmac.finalize() == mac   # 生成MAC mac = generate\_mac(key, message) print(f'Generated MAC: {b64encode(mac).decode()}')  # 验证MAC is\_valid = verify\_mac(key, message, mac) print(f'MAC valid: {is\_valid}')  # 尝试验证一个错误的MAC invalid\_mac = b'\x00' \* len(mac) # 一个与生成的MAC不同的值 is\_invalid = verify\_mac(key, message, invalid\_mac) print(f'Invalid MAC valid: {is\_invalid}')  *from cryptography.hazmat.primitives.asymmetric import rsa, padding from cryptography.hazmat.primitives import hashes, serialization from cryptography.hazmat.primitives.serialization import Encoding from cryptography.hazmat.primitives.asymmetric.utils import (  decode\_dss\_signature, encode\_dss\_signature )   # 生成RSA密钥对 def generate\_keys():  private\_key = rsa.generate\_private\_key(  public\_exponent=65537,  key\_size=2048,  )  public\_key = private\_key.public\_key()  return private\_key, public\_key   # 对消息进行签名 def sign\_message(private\_key, message):  signature = private\_key.sign(  message,  padding.PSS(  mgf=padding.MGF1(hashes.SHA256()),  salt\_length=padding.PSS.MAX\_LENGTH  ),  hashes.SHA256()  )  return signature   # 验证签名 def verify\_signature(public\_key, message, signature):  try:  public\_key.verify(  signature,  message,  padding.PSS(  mgf=padding.MGF1(hashes.SHA256()),  salt\_length=padding.PSS.MAX\_LENGTH  ),  hashes.SHA256()  )  return True  except Exception as e:  return False   # 示例消息   message = b'This is a secret message.'  # 生成密钥对 private\_key, public\_key = generate\_keys()  # 保存密钥到文件（可选） with open("private\_key.pem", "wb") as f:  f.write(private\_key.private\_bytes(  encoding=Encoding.PEM,  format=serialization.PrivateFormat.PKCS8,  encryption\_algorithm=serialization.NoEncryption()  ))  with open("public\_key.pem", "wb") as f:  f.write(public\_key.public\_bytes(  encoding=Encoding.PEM,  format=serialization.PublicFormat.SubjectPublicKeyInfo  ))  # 对消息进行签名 signature = sign\_message(private\_key, message) print(f"Signature: {signature.hex()}")  # 验证签名 is\_valid = verify\_signature(public\_key, message, signature) print(f"Signature valid: {is\_valid}")*  *from cryptography.hazmat.primitives.asymmetric import ec from cryptography.hazmat.primitives import hashes from cryptography.hazmat.primitives.asymmetric import utils from cryptography.exceptions import InvalidSignature  # 生成椭圆曲线私钥和公钥对 private\_key = ec.generate\_private\_key(ec.SECP384R1()) public\_key = private\_key.public\_key()  # 要签名的消息 message = b"This is a test message for ECDSA signing."  # 签名消息 signature = private\_key.sign(  message,  ec.ECDSA(hashes.SHA256()) )  print("Signature:", signature.hex()) # 验证签名 try:  public\_key.verify(  signature,  message,  ec.ECDSA(hashes.SHA256())  )  print("The signature is valid.") except InvalidSignature:  print("The signature is invalid.")   # 注意：这里没有保存私钥和公钥到文件的代码*  *'''MAC''' from Crypto.Hash import HMAC, SHA256 from Crypto.Random import get\_random\_bytes import binascii  # 生成一个随机密钥 key = get\_random\_bytes(16)  # 要认证的消息 message = b'This is a secret message.'  # 生成HMAC对象 hmac\_obj = HMAC.new(key, digestmod=SHA256) hmac\_obj.update(message)  # 获取MAC mac = hmac\_obj.hexdigest()  print(f'MAC: {mac}')  # 验证MAC hmac\_obj\_verify = HMAC.new(key, digestmod=SHA256) hmac\_obj\_verify.update(message) mac\_verify = hmac\_obj\_verify.hexdigest()  if mac == mac\_verify:  print('MAC verification succeeded.') else:  print('MAC verification failed.')  '''Signature''' from Crypto.PublicKey import RSA from Crypto.Signature import pkcs1\_15 from Crypto.Hash import SHA256 import binascii  # 生成RSA密钥对 key = RSA.generate(2048) private\_key = key.export\_key() public\_key = key.publickey().export\_key()  # 要签名的消息 message = b'This is a message for digital signature.'  # 生成哈希对象 hash\_obj = SHA256.new(message)  # 生成签名 signature = pkcs1\_15.new(key).sign(hash\_obj)  print(f'Signature: {binascii.hexlify(signature).decode()}')  # 验证签名 try:  pkcs1\_15.new(key.publickey()).verify(hash\_obj, signature)  print('Signature verification succeeded.') except (ValueError, TypeError):  print('Signature verification failed.')* | | | |
| 结论分析与体会：  通过本次实验，我深入了解了消息认证的重要性以及其在保障通信安全中的关键作用。实验涵盖了基于对称密码的MAC（消息认证码）和非对称密码的数字签名两种实现方式，让我对这两种消息认证技术有了更为直观和深刻的理解。  在利用cryptography库中的primitives.hashes和hmac模块实现给定消息的认证码生成及验证时，我深刻体会到了HMAC（基于哈希的消息认证码）的简便性和高效性。通过为消息附加一个由发送者私钥和消息内容共同决定的哈希值，接收者可以轻松地验证消息的完整性和真实性。  随后，我尝试使用cmac和algorithms.AES来生成和验证消息的认证码。CMAC（Cipher-based Message Authentication Code）是一种基于分组密码的消息认证码，它利用AES等对称加密算法来确保消息的机密性和完整性。在实验过程中，我深刻感受到了CMAC在抵御伪造攻击和篡改攻击方面的强大能力。  在探索数字签名时，我分别使用了RSA和椭圆曲线算法（EC）。RSA数字签名利用了大整数分解的数学难题，通过私钥对消息进行签名，公钥进行验证，从而确保了消息的不可否认性和完整性。而椭圆曲线数字签名则利用了椭圆曲线离散对数问题的难解性，相比RSA具有更高的安全性和更小的密钥长度。这两种数字签名方式都让我深刻体会到了非对称密码在消息认证中的独特优势。  此外，我还尝试了使用Crypto库中的对应功能模块来编程实现消息认证码和数字签名的生成和验证。虽然这个库的使用相对复杂一些，但通过不断的尝试和调试，我逐渐掌握了其使用方法，并成功实现了消息认证的功能。  在熟悉代码5-6、5-7、5-8的过程中，我对数字证书的内容结构、签发方式和作用有了更为全面的了解。数字证书作为网络通信中的“身份证”，不仅包含了证书持有者的身份信息、公钥等关键信息，还通过数字签名确保了这些信息的真实性和完整性。通过数字证书的签发和验证过程，我们可以有效地防止中间人攻击和身份伪造等安全问题。  总的来说，本次实验让我对消息认证和数字证书有了更为深入和全面的了解。我不仅掌握了基于对称密码和非对称密码的消息认证技术，还了解了数字证书的结构和作用。这些知识和技能对于我在未来的网络通信安全领域的学习和工作中都将起到重要的指导作用。 | | | |

注：命名”学号姓名.docx”，例如：“2006999孔明.docx”