University of Miami

Crypto Lab 2

ECE 576

Nigel John
10-11-2019

Objectives:

1. To understand and use symmetric and asymmetric key cryptography
2. To securely exchange a file over an unsecure channel

Task 0: Install python (v3), python IDE, python cryptography

For this lab, you are going to need to have python installed on your laptop. If you already have this then you can skip this step.

1. Download the latest python (v3) from www.python.org for your operating system
    a. Your SEEDs systems have python 2 but not python 3
    b. Your Kali has both, but you may need to update
2. Follow the installation instructions for your system
    a. See: https://wiki.python.org/moin/BeginnersGuide
3. For programming information:
    a. https://wiki.python.org/moin/HowToEditPythonCode
4. Install the python cryptography module
    a. Open a Terminal window (or Command shell)
    b. Type: pip install cryptography
        i. check https://cryptography.io for installation issues
5. Download and install an IDE:
    a. IDLE: python ships with a standard IDE, IDLE, written in pure python
    b. PyCharm: www.jetbrains.com/pycharm
    c. Eclipse with PyDev: www.eclipse.org (my personal preference)
    d. Xcode on the Mac also supports python
6. Familiarize yourself with the platform
7. For this lab you also need openSSL for the backend to get access to the PK algorithms
8. The python documentation is at:
    a. Main documents: https://docs.python.org/3/
    b. Tutorial: https://docs.python.org/3/tutorial/index.html

Task 1: Create and use Message Digests (Hashes)

Message Digests are basic hashes. These are created in much the same way as the Cipher objects. They do not have keys associated with them, so they only provide based integrity checks, but are easily bypassed.

Task 1.1

1. **Create a python file**
2. **Import**
    a. ***from cryptography.hazmat.backends import default_backend***
    b. ***from cryptography.hazmat.primitives import hashes***
3. **Set the backend**
    a. ***backend = default_backend()***
4. **Read in a line from the user into a bytearray data**
5. **Create a hash object**
    a. ***myhash = hashes.SHA256()***
    b. ***hasher = hashes.Hash(myhash, backend)***
6. **Add data to the message digest**
    a. **Keep calling: *hasher.update(<data>)* as you read in data**
7. **Get the digest by calling finalize on the hash object**
    a. ***digest = hasher.finalize()***
8. **Output the original data and the message digest**
9. **Try for different size messages and with slight differences in the message**

Task 1.2

1. **Modify your program from Task 1.1 to use different message digests**
    a. **Such as: MD5, SHA1, SHA256, SHA-384, SHA-512**

Task 2: Create and use RSA public/private key pair

Now you will create a RSA key pair.

1. **Create a python file**
2. **Import**
    a. ***from cryptography.hazmat.backends import default_backend***
    b. ***from cryptography.hazmat.primitives.asymmetric import rsa***
    c. ***from cryptography.hazmat.primitives import serialization***
3. **Set the backend**
    a. ***backend = default_backend()***
4. **Generate a private key/public key pair for RSA**
    a. ***private_key = rsa.generate_private_key(***
    b. ***public_exponent=65537,***
    c. ***key_size=2048,***
    d. ***backend=backend)***
        i. **The exponent is the public key value**
        ii. **The key size is the number of bits of the key**
        iii. **Use the openSSL backend**
5. **Get the public key part**
    a. ***public_key = private_key.public_key()***
6. **Encode the keys for serialization and saving, first the private key (which should be kept encrypted for security)**
    a. ***password = 'hello'***
    b. ***pem_kr = private_key.private_bytes(***
    c. ***encoding=serialization.Encoding.PEM,***
    d. ***format=serialization.PrivateFormat.PKCS8,***
    e. ***encryption_algorithm=serialization.BestAvailableEncryption(password.encode())***
        i. **Encoding is PEM (Privacy Enhanced Mail), base64 encoded DER (Distinguished Encoding Rules) data**
        ii. **Format is PKCS#8 - private key serialization format (see RFC5208)**
        iii. **Encryption is provided by a built-in algorithm**
7. **Now the public key (is not encrypted)**
    a. ***pem_ku = public_key.public_bytes(***
    b. ***encoding=serialization.Encoding.PEM,***
    c. ***format=serialization.PublicFormat.SubjectPublicKeyInfo)***
8. **Save the results to two separate files (these will act as your keyring)**
    a. **Save pem_kr to kr.pem**
    b. **Save pem_ku to ku.pem**
9. **Make sure that you can reload these keys**
    a. ***with open(kr_fname,'rb') as file:***
    b. ***private_key = serialization.load_pem_private_key(***
    c. ***data=file.read(),***
    d. ***password=password.encode(),***
    e. ***backend=backend)***
    f. ***with open(ku_fname,'rb') as file:***
    g. ***public_key = serialization.load_pem_public_key(***
    h. ***data=file.read(),***
    i. ***backend=backend)***

Task 3: Combine the message digest with the signature to create a signed document

Combine the code from Task 1 and Task 2 to create a signed message digest that acts as a signature.

Task 3.1

1. **Create a new python file**
2. **Import**
    a. ***from cryptography.hazmat.backends import default_backend***
    b. ***from cryptography.hazmat.primitives.asymmetric import rsa***
    c. ***from cryptography.hazmat.primitives import serialization***
    d. ***from cryptography.hazmat.primitives import hashes***
    e. ***from cryptography.hazmat.primitives.asymmetric import padding***
    f. ***from cryptography.hazmat.primitives.asymmetric import utils***
    g. ***from encodings.base64_codec import base64_encode***
3. **Load and hash the data to be signed, from Task 1**
4. **Load the private key from Task 2**
5. **The hashed data needs to be padded so that it fits properly with the signature (rsa encrypt with private key) algorithm, use the PSS padding algorithm designed for padding signatures. Note the PKCS#1v1.5 may also be used, but has some security implications that makes it not recommended**
    a. ***pad = padding.PSS(***
    b. ***mgf=padding.MGF1(hashes.SHA256()),***
    c. ***salt_length=padding.PSS.MAX_LENGTH***
    d. ***)***
6. **Now sign the data with the private key, giving it the digest you computed earlier and the pad object you just created**
    a. ***sig = private_key.sign(***
    b. ***data=digest,***
    c. ***padding=pad,***
    d. ***algorithm=utils.Prehashed(myhash))***
        i. **Note the use of the Prehashed into which you pass the hash object you created**
7. **Save the signature to a file with a .sig extension with a PEM format:**
    a. **-----BEGIN SIGNATURE-----**
    b. **<base64 encoded signature>**
    c. **-----END SIGNATURE-----**

Task 3.2

Now we need to verify the signature.

1. **Create a new python file**
2. **Import**
    a. ***from cryptography.hazmat.backends import default_backend***
    b. ***from cryptography.hazmat.primitives.asymmetric import rsa***
    c. ***from cryptography.hazmat.primitives import serialization***
    d. ***from cryptography.hazmat.primitives import hashes***
    e. ***from cryptography.hazmat.primitives.asymmetric import padding***
    f. ***from cryptography.hazmat.primitives.asymmetric import utils***
    g. ***from encodings.base64_codec import base64_encode***
3. **Load and hash the data to be signed, from Task 1**
4. **Load the public key from Task 2**
5. **Load the signature from the signature file**
    a. **Remember to remove the start and end lines**
    b. **Also decode the data from base64 back to raw bytes**
6. **The hashed data was padded so you need to unpad using the same padding algorithm, create it the same way as in Task3.1**
7. **Verify the signature**
    a. ***public_key.verify(***
    b. ***signature=sig,***
    c. ***data=digest,***
    d. ***padding=pad,***
    e. ***algorithm=utils.Prehashed(myhash))***
        i. **The verify will raise a cryotpgraphy.exceptions.InvalidSignature if the signature is not valid**
        ii. **You can put the verify into a try to handle the exception**
            1. **try:**
                a. **<code to try>**
            2. **except <exception to catch>:**
                a. **<code when exception happens>**
                b. **<except clauses may be repeated for more exceptions>**
                c. **<or collected into one if effect is same for all>**
            3. **else:**
                a. **<code to execute is no exception happened>**

Task 4: Combine the symmetric encryption of a file with a signed message digest

Combine your file encryption code from Crypto Lab 1 with the code from Task 3 to create an encrypted and signed file. Note it's better to encrypt and then sign as you can verify before having to decrypt.

Report

1. Statement of what was done for each task
2. All source code for each task
3. Screen shots of all operations for each task

Reference:

Python reference: https://docs.python.org/3/
Cryptography module reference: https://cryptography.io/en/latest/