Programming using Public-Key Cryptography APIs

This Video Covers:

- Key Generation
- Encryption and Decryption
- Digital Signature

Programming using Public-Key Cryptography APIs

- Languages (e.g., Python, Java, C/C++) have well-developed libraries that implement the low-level cryptographic primitives for public-key operations

- Python:
  - No *built-in* cryptographic library
  - Use Python packages (e.g., **PyCryptodome**)

Public-Key Cryptography APIs: ***Key Generation***

Python Example:

Use Python Crypto APIs to generate a RSA key and save it to a file

```
#!/usr/bin/python3

from Crypto.PublicKey import RSA

key = RSA.generate(2048) # (1) generate a 2048-bit RSA key
pem = key.export_key(format='PEM', passphrase='dees') # (2) export key() API serializes
                                                      # the key using the ASN.1 structure
f = open('private.pem','wb')
f.write(pem)
f.close()

pub = key.publickey() # (3) extract public-key component
pub_pem = pub.export_key(format='PEM')
f = open('public.pem','wb')
f.write(pub_pem)
f.close()
```

key_gen.py

Public-Key Cryptography APIs: ***Encryption***

Python Example:

To encrypt a message using public keys, we need to decide what padding scheme to use

```
#!/usr/bin/python3

from Crypto.Cipher import PKCS1_OAEP
from Crypto.PublicKey import RSA

message = b'A secret message!\n'

key = RSA.importKey(open('public.pem').read()) # import pub key from file
cipher = PKCS1_OAEP.new(key) # create a cipher object using pub key
ciphertext = cipher.encrypt(message)
f = open('ciphertext.bin','wb')
f.write(ciphertext)
f.close()
```

NOTE: For better security, it is recommended that OAEP is used

encrypt.py

Public-Key Cryptography APIs: ***Decryption***

Python Example:

Uses the private key and the decrypt() API

```
#!/usr/bin/python3

from Crypto.Cipher import PKCS1_OAEP
from Crypto.PublicKey import RSA

ciphertext = open('ciphertext.bin', 'rb').read()

prikey_pem = open('private.pem').read() # import private key from file ...
prikey = RSA.importKey(prikey_pem, passphrase='dees') # ... w/ passphrase
cipher = PKCS1_OAEP.new(prikey) # create a cipher object
message = cipher.decrypt(ciphertext)
print(message)
```

decrypt.py

Public-Key Cryptography APIs: ***Digital Signatures***

- In Python code, one can use PyCryptodome library's `Crypto.Signature` package

- Four supported digital signature algorithms:
  - RSASSA-PKCS1-v1_5
  - RSASSA-PSS ← we will see an example with RSASSA-PSS
  - DSA

Public-Key Cryptography APIs: ***Digital Signatures Using PSS***

- **Probabilistic Signature Scheme (PSS)** is a cryptographic signature scheme designed by Mihir Bellare and Phillip Rogaway
- RSA-PSS is standardized as part of PKCS#1 v2.1
- Sign a message in combination with some random input.
- For the same input:
  - two signatures are different
  - both can be used to verify

Public-Key Cryptography APIs: ***Digital Signatures Using PSS*** *(cont.)*

```
#!/usr/bin/python3

from Crypto.Signature import pss
from Crypto.Hash import SHA256
from Crypto.PublicKey import RSA

message = b'An important message'
key_pem = open('private.pem').read()
key = RSA.import_key(key_pem, passphrase='dees')
h = SHA256.new(message)
signer = pss.new(key) # create a signature object
signature = signer.sign(h) # gen. the sig. for the hash of a message
open('signature.bin', 'wb').write(signature)
```

sign.py

Public-Key Cryptography APIs: ***Digital Signatures Using PSS*** *(cont.)*

```
#!/usr/bin/python3

from Crypto.Signature import pss
from Crypto.Hash import SHA256
from Crypto.PublicKey import RSA

message = b'An important message'
signature= open('signature.bin', 'rb').read()
key = RSA.import_key(open('public.pem').read())
h = SHA256.new(message)
verifier = pss.new(key)
try:
    verifier.verify(h, signature)
    print("The signature is valid.")
except (ValueError, TypeError):
    print("The signature is NOT valid.")
```

verify.py