Week 12 Homework due on Dec 7, 18:59 Hour

Group 5

Wong Ann Yi (1004000)
Liu Bowen (1004028)
Tan Chin Leong Leonard (1004041)

Exercise 1

Incorporate your digital certificate framework (this week's classwork) to your key negotiation protocol (Week 11 classwork). Then incorporate both to your secure channel implementation (Week 9 homework). More specifically, in the final system:

a) Alice and Bob trust a CA (i.e., Alice and Bob have the CA's certificate).
b) This CA issues certificates for Alice and Bob respectively.
c) Alice initiates a connection with Bob, starting an authenticated key negotiation. (She needs to send her certificate which is then validated by Bob.)
d) Bob authenticates the negotiation. (He also needs to send his certificate to Alice.)
e) After a shared key is established, the secure channel can be initiated.

Answers:

When Alice and Bob wants to communicate securely, they would both generates their public/private key pairs. They would take their public keys to the CA to have them signed and certified. Both Alice and Bob would exchange their public keys and verify them by the certificates. They would then conduct the key negotiation protocol to set up a session key which will be used to set up a secure channel to exchange data. This protocol will utilize the Diffie-Hellman key exchange protocol (final version) and the associated parameters to generate a session key. This key will be used to establish the secure channel.

When establishing a secure channel, the session secret key with a 256-bit value is generated in the above key negotiation process for Alice and Bob only. The secure channel design is made up of three components: message numbering, authentication, and encryption. The message number will help Bob keep track of the messages from Alice and to prevent replay attack from Eve. The message number is also a source for IVs for the encryption algorithm. For authentication, we will use HMAC-SHA-256 to generate the full 256-bit results. For encryption, we will use AES in CBC mode with the nonce value generated by the message number.

Using the codes written in week 12 classwork, week 11 classwork and week 9 homework, we develop a full set of Python codes that allows Alice and Bob to generate public and private keys, have their public keys signed by CA, set up a session key using the Key Negotiation protocol and using it to establish a secure channel for communication between them. The codes and the outputs are listed below.

```
from Cryptodome.Cipher import AES
from Crypto.Hash import SHA256
from Crypto.Signature import PKCS1_v1_5
from Crypto.PublicKey import RSA
import time
import math
import random
import hashlib
import binascii
import hmac
from Cryptodome.Protocol import KDF

class KeyNego(object):
   def __init__(self):
      self.key = RSA.generate(1024)

   def isPrime(self, n):
           if n <=1: return False
           i = 2
           while i*i <= n:
                      if n%i == 0: return False
                      i += 1
           return True

   def generatePrime(self, keysize):
      while True:
         num = random.randrange(2**(keysize-1), 2**(keysize))
         if self.isPrime(num):
            return num

   def choose_gpq(self, Sa, Sb):
      alfa = 2
      g = 0
      s = max(Sa, Sb)
      if s > 2*Sb:
         return False
      while(True):
         p = self.generatePrime(s)
         if math.log(p,2) < s-1:
            continue
         q = (p-1)/2
         if not self.isPrime(q):
            continue
         while(True):
            alfa = random.randint(2, p-2)
            g = alfa**2 % p
            if g == 1:
               continue
```

```
            if g == p-1:
                continue
            break
        b = random.randint(1, q-1)
        B = g**b % p
        break
    message = g+p+q+B
    key = RSA.generate(1024)
    publickey = key.publickey()
    h = SHA256.new(hex(message))
    signature = PKCS1_v1_5.new(key).sign(h)
    return g, p, q, b, B, publickey, signature

def receiveFromAlice(self, Sa, g, p, q, B, publickey, signature):
    message = g+p+q+B
    hash_message = SHA256.new(hex(message))
    if PKCS1_v1_5.new(publickey).verify(hash_message, signature) == False:
        print 'error'
    if ((Sa-1) < math.log(p,2)) & (math.log(p,2) < 2*Sa):
        if not self.isPrime(q):
            print 'error'
        if not self.isPrime(p):
            print 'error'
        if 2*q != (p-1):
            print 'error'
        if g == 1:
            print 'error'
        if g == p-1:
            print 'error'
        if B == 1:
            print 'error'
        if B**q % p != 1:
            print 'error'
        a = random.randint(1, q-1)
        A = g**a % p
        K_prime = B**a % p
        h = SHA256.new()
        h.update(hex(K_prime))
        K = h.hexdigest()
        message1 = A
        key_new = RSA.generate(1024)
        publickey_new = key_new.publickey()
        hash_new = SHA256.new(hex(message1))
        signature_new = PKCS1_v1_5.new(key_new).sign(hash_new)
        return a, A, K_prime, K, publickey_new, signature_new
    else:
        print 'error'
```

```
    def bobReceive(self, A, g, p, q, b, publickey_new, signature_new):
        message = A
        hash_message = SHA256.new(hex(message))
        if PKCS1_v1_5.new(publickey_new).verify(hash_message, signature_new) == False:
            print 'error'
        if A == 1:
            print 'error'
        if A**q % p != 1:
            print 'error'
        K_prime = A**b % p
        h = SHA256.new()
        h.update(hex(K_prime))
        K = h.hexdigest()
        return K_prime, K

class RootCA(object):
    def __init__(self):
        self.key = RSA.generate(1024)
        self.publicKey = self.key.publickey()

    def generateCertiChain(self, name, public_key):
        owner = name
        issuer = 'RootCA'
        not_before = int(time.time())
        not_after = int(time.time() + 10*365*24*60*60)
        publicKey = public_key
        message = owner+issuer+str(not_before)+str(not_after)
        hash_message = SHA256.new(message)
        signature = PKCS1_v1_5.new(self.key).sign(hash_message)
        certificate = []
        certificate.append(owner)
        certificate.append(issuer)
        certificate.append(not_before)
        certificate.append(not_after)
        certificate.append(publicKey)
        certificate.append(signature)
        return certificate

    def applyCertificateChain(self, name, message, public_key, signature):
        hash_message = SHA256.new(message);
        verified = PKCS1_v1_5.new(public_key).verify(hash_message, signature)
        if verified == True:
            return self.generateCertiChain(name, public_key);

    def getPublicKey(self):
        return self.publicKey;

class Alice(object):
```

```
    def __init__(self):
        self.name = 'Alice'
        self.key = RSA.generate(1024)
        self.publicKey = self.key.publickey()
        self.CAdic = {}
        self.Sa = 8;
        self.share_key = '';
        self.certificate = None;
        self.sender_counter = 0;
        self.receiver_counter = 0;
        self.encry_key = (KDF.HKDF(self.share_key, salt=None, key_len=32, hashmod=SHA256,
num_keys=2, context=None))[0]
        self.auth_key = (KDF.HKDF(self.share_key, salt=None, key_len=32, hashmod=SHA256,
num_keys=2, context=None))[1]

    def getPKIKey(self, name, PKI_object):
         key = PKI_object.getPublicKey();
         self.CAdic[name] = key;

    def applyCertificateChain(self, message, PKI_object):
        hash_message = SHA256.new(message)
        signature = PKCS1_v1_5.new(self.key).sign(hash_message)
        certificate = PKI_object.applyCertificateChain(self.name, message, self.publicKey,
signature)
        self.certificate = certificate
        return certificate

    def sendToBob(self, certificate, send_objetc, nego_object):
        sending = []
        N = random.randint(0, 2**256-1)
        sending.append(self.Sa)
        sending.append(N)
        sending.append(certificate)
        print '======== step 4.1: Alice sending Sa, N, certificate =========='
        print 'Sa:', self.Sa
        print 'N:', N
        return send_objetc.BobVerify(sending, nego_object, self)

    def receiveBobNego(self, g, p, q, b, B, publickey, signature, nego_object, bob_object,
certificate):
        print '======== step 4.3: Alice verify certificate(Bob) ============='
        verified = self.AliceVerify(certificate)
        if verified == True:
            a, A, K_prime, K, publickey_new, signature_new =
nego_object.receiveFromAlice(self.Sa, g, p, q, B, publickey, signature)
            self.share_key = K
            print '======== Alice sends A, sig(Alice) to Bob ===================='
            print 'A:', A
```

```
        print '======== Alice generate shared_key =========================='
        bob_object.generateShareKey(A, g, p, q, b, publickey_new, signature_new, 
nego_object);

    def getShareKey(self):
        return self.share_key;

    def AliceVerify(self, certificate):
        if time.time() < certificate[2] or time.time() > certificate[3]:
            return False;
        message = certificate[0]+certificate[1]+str(certificate[2])+str(certificate[3]);
        hash_message = SHA256.new(message);
        verified = PKCS1_v1_5.new(self.CAdic[certificate[1]]).verify(hash_message, certificate[5])
        if verified == False:
            return False;
        else:
            return True;

    def send(self, msg):
        print '======== step 5.1: Alice send (First meg from Alice!) ========'
        IV = hashlib.sha256(format(self.sender_counter, 'x')).hexdigest()[:16]
        cipher = AES.new(self.encry_key, AES.MODE_CBC, IV)
        if len(msg) % 16 == 0:
            message = msg
        else:
            message = msg.encode('hex') + '80' + (16 - len(msg) % 16 - 1)*'00'
        ciphertext = binascii.hexlify(cipher.encrypt(message))
        authtext = (hmac.new(self.auth_key, ciphertext, hashlib.sha256).hexdigest())
        counter_len = len((format(self.sender_counter, 'x')))
        if counter_len < 4*2:
            counter = (4*2 - counter_len)*'0' + (format(self.sender_counter, 'x'))
        else:
            counter = counter_len
        protected_msg = counter + ciphertext + authtext
        self.sender_counter += 1
        print '======== Whole sending message: =============================='
        print 'counter:', counter
        print 'ciphertext:', (ciphertext)
        print 'authtext:', (authtext)
        return protected_msg

    def receive(self, protected_msg):
        print '========= step 5.4: Alice decrypte meg from Bob =============='
        size = len(protected_msg)
        authtext =  protected_msg[size-64:]
        ciphertext = (protected_msg[8:size-64])
        counter = int(protected_msg[0:8], 16)
        verify_authtext = hmac.new(self.auth_key, (ciphertext), hashlib.sha256).hexdigest()
```

```
    if (authtext == verify_authtext):
      IV = hashlib.sha256(format(counter, 'x')).hexdigest()[:16]
      cipher = AES.new(self.encry_key, AES.MODE_CBC, IV)
      msg = cipher.decrypt((ciphertext).decode('hex'))
      msg_size = len(msg)
      for i in range(msg_size / 2):
        if msg[msg_size-2:] == '00':
          msg = msg[0:msg_size-2]
          msg_size -= 2
          continue;
        if msg[msg_size-2:] == '80':
          msg = msg[0:msg_size-2]
          msg_size -= 2
          break;
      self.receiver_counter += 1
      print 'Msg to Bob:', binascii.unhexlify(msg)
    else:
      print 'authenticate failed!'

class Bob(object):
  def __init__(self):
    self.name = 'Bob'
    self.key = RSA.generate(1024)
    self.publicKey = self.key.publickey();
    self.CAdic = {}
    self.Sb = 10;
    self.share_key = '';
    self.certificate = None;
    self.sender_counter = 0;
    self.receiver_counter = 0;
    self.encry_key = (KDF.HKDF(self.share_key, salt=None, key_len=32, hashmod=SHA256,
num_keys=2, context=None))[0]
    self.auth_key = (KDF.HKDF(self.share_key, salt=None, key_len=32, hashmod=SHA256,
num_keys=2, context=None))[1]

  def getPKIKey(self, name, PKI_object):
    key = PKI_object.getPublicKey();
    self.CAdic[name] = key;

  def applyCertificateChain(self, message, PKI_object):
    hash_message = SHA256.new(message)
    signature = PKCS1_v1_5.new(self.key).sign(hash_message)
    certificate = PKI_object.applyCertificateChain(self.name, message, self.publicKey,
signature)
    self.certificate = certificate
    return certificate

  def sendToAlice(self, certificate, send_objetc):
```

```
        return send_objetc.AliceVerify(certificate)

    def BobSendNego(self, Sa, nego_object, alice_object):
        g, p, q, b, B, publickey, signature = nego_object.choose_gpq(Sa, self.Sb)
        print '======== then Bob sends (g,p,q), B, sig(Bob), certificate(Bob) ==='
        print '(g,p,q):', g,p,q
        print 'B:', B
        alice_object.receiveBobNego(g, p, q, b, B, publickey, signature, nego_object, self,
self.certificate)

    def generateShareKey(self, A, g, p, q, b, publickey_new, signature_new, nego_object):
        print '======== step 4.4: Bob verify and generate shared_key ========'
        K_prime, K = nego_object.bobReceive(A, g, p, q, b, publickey_new, signature_new)
        self.share_key = K

    def getShareKey(self):
        return self.share_key;

    def BobVerify(self, certificate, nego_object, alice_object):
        if time.time() < certificate[2][2] or time.time() > certificate[2][3]:
            return False;
        message = certificate[2][0]+certificate[2][1]+str(certificate[2][2])+str(certificate[2][3]);
        hash_message = SHA256.new(message);
        verified = PKCS1_v1_5.new(self.CAdic[certificate[2][1]]).verify(hash_message,
certificate[2][5])
        print '======== step 4.2: Bob verify certificate(Alice) ============='
        if verified == False:
            return False;
        else:
            self.BobSendNego(certificate[0], nego_object, alice_object)
            return True;

    def send(self, msg):
        print '========= step 5.3: Bob send (First meg from Bob!) ==========='
        IV = hashlib.sha256(format(self.sender_counter, 'x')).hexdigest()[:16]
        cipher = AES.new(self.encry_key, AES.MODE_CBC, IV)
        if len(msg) % 16 == 0:
            message = msg
        else:
            message = msg.encode('hex') + '80' + (16 - len(msg) % 16 - 1)*'00'
        ciphertext = binascii.hexlify(cipher.encrypt(message))
        authtext = (hmac.new(self.auth_key, ciphertext, hashlib.sha256).hexdigest())
        counter_len = len((format(self.sender_counter, 'x')))
        if counter_len < 4*2:
            counter = (4*2 - counter_len)*'0' + (format(self.sender_counter, 'x'))
        else:
            counter = counter_len
        protected_msg = counter + ciphertext + authtext
```

```
        self.sender_counter += 1
        print '======== Whole sending message: =============================='
        print 'counter:', counter
        print 'ciphertext:', (ciphertext)
        print 'authtext:', (authtext)
        return protected_msg

    def receive(self, protected_msg):
        print '======= step 5.2: Bob decrypte meg from Alice =============='
        size = len(protected_msg)
        authtext =  protected_msg[size-64:]
        ciphertext = (protected_msg[8:size-64])
        counter = int(protected_msg[0:8], 16)
        verify_authtext = hmac.new(self.auth_key, (ciphertext), hashlib.sha256).hexdigest()
        if (authtext == verify_authtext):
            IV = hashlib.sha256(format(counter, 'x')).hexdigest()[:16]
            cipher = AES.new(self.encry_key, AES.MODE_CBC, IV)
            msg = cipher.decrypt((ciphertext).decode('hex'))
            msg_size = len(msg)
            for i in range(msg_size / 2):
                if msg[msg_size-2:] == '00':
                    msg = msg[0:msg_size-2]
                    msg_size -= 2
                    continue;
                if msg[msg_size-2:] == '80':
                    msg = msg[0:msg_size-2]
                    msg_size -= 2
                    break;
            self.receiver_counter += 1
            print 'Msg to Alice:', binascii.unhexlify(msg)
        else:
            print 'authenticate failed!'

print '======= Step 1: Initialize class ==========================='
alice = Alice()
bob = Bob()
rootca = RootCA()
keyNego = KeyNego()

print '\n======= Step 2: Alice, Bob get RootCA public key ==========='
alice.getPKIKey('RootCA', rootca)
bob.getPKIKey('RootCA', rootca)

print '\n======= Step 3: Alice, Bob get certificate from RootCA ======'
aliceCeriFromRoot = alice.applyCertificateChain('I am Alice!', rootca)
print 'Alice certificate:\n', aliceCeriFromRoot
bobCeriFromRoot = bob.applyCertificateChain('I am Bob!', rootca)
print 'Bob certificate:\n', bobCeriFromRoot
```

```
print '\n======== Step 4: Alice initiates connection with Bob ========='
verified = alice.sendToBob(aliceCeriFromRoot, bob, keyNego)
alice_share_key = alice.getShareKey()
bob_share_key = bob.getShareKey()
print 'shared_key of Alice is:', alice_share_key
print 'shared_key of Bob is:', bob_share_key

print '\n======== Step 5: After shared_key is established ============='
msg1 = alice.send("First meg from Alice!")
bob.receive(msg1)
msg3 = bob.send("First meg from Bob!")
alice.receive(msg3)
print '\n======== All steps finish! ===================================='
```

The outputs are listed below.

```
======== Step 1: Initialize class ============================

======== Step 2: Alice, Bob get RootCA public key ============

======== Step 3: Alice, Bob get certificate from RootCA ======

Alice certificate:

['Alice', 'RootCA', 1543929578, 1859289578, <_RSAobj @0x1c193df290
n(1024),e>,
"ZK_\x8fn\xc7k\xea2\x85O\xfe\x10bwm\xc1q\x84\x05\x08D$\x1c\xf2\xb2[5{\
xcd\x05\x11\xc1\xcb\xa0\x87\xedg\x0e\x86 >\x97\x98\xb9\x02\xc9\x95o\x8
3\x04\x18\x14\x18\xa6,\tO`#\x82&Q\x00\xe2\xf61H\x81\xe4L,]\xaeG*\xb1G\
xdd\x96cGK\x10\x01Q&|`Y\xfb^\x8e\x99\xfa(\x08\x15\xdd\x92'\xc9\x03\xc1
\xa8c\x7f\xe0v\xdc\xb9\x05\x18\x19\x8b\x19\x1aR@\x08\xb2\xe1\xa1\x85[\
xc8i."]

Bob certificate:

['Bob', 'RootCA', 1543929578, 1859289578, <_RSAobj @0x1c18c0ba28
n(1024),e>,
"\x18\x91{Q\t+V\x8c\xcfy2Kk<\x03\x03Rr\xfdh\xc7'\xfb\xe3RUf\r\x19\x82L
q\xde\x8f\x92'\x02\xec\xd3\x13\x02\x06c\xd8\xac\xe9Q\xc3\xa2oj\xb4\xcc
\x01\xf4\x84l_C\x0f\xa8\xf2\xfd\xb8\x9c\x93\xcd\xb5\xac\xbe\x12\xceY\x
eaMw\x02D\xea\x0315Ly\xbbKi\x07[\x8b\xa7(I\xcc[\xb2y\x83F\xb6\xb0\x80\
xdf\xf4\xce=\xa7\xee3o\xd3qb{\x7ftH\xc2\x15\xd2\x94\xd85\x00\xba\x06\x
1f\x18"]



======== Step 4: Alice initiates connection with Bob =========
```

```
======== step 4.1: Alice sending Sa, N, certificate ==========

Sa: 8

N:
77090274570612698066897547250776542792460753417077377995569023629059508795457

======== step 4.2: Bob verify certificate(Alice) =============

======== then Bob sends (g,p,q), B, sig(Bob), certificate(Bob) ===

(g,p,q): 189 563 281

B: 197

======== step 4.3: Alice verify certificate(Bob) =============

======== Alice sends A, sig(Alice) to Bob ====================

A: 183

======== Alice generate shared_key ===========================

======== step 4.4: Bob verify and generate shared_key ========

shared_key of Alice is:
1bd69d78f424afc1d10edb599ad44be71d0b99167c51dc9f3af087a79dacb059

shared_key of Bob is:
1bd69d78f424afc1d10edb599ad44be71d0b99167c51dc9f3af087a79dacb059


======== Step 5: After shared_key is established =============

======== step 5.1: Alice send (First meg from Alice!) ========

======== Whole sending message: ==============================

counter: 00000000

ciphertext:
eafa6ecc33c10332646e4a918e8eb3e52dd220e54540f384e635210bf30dbb90432fba5d824e3d33a2b53604676700d5755b12d36064417438ae12ae1fb4e059

authtext:
f80566f5b717d892dfc4a48e32109adc5f850e36c7a0ea3916bcdc51ea0ff2d7
```

```
======== step 5.2: Bob decrypte meg from Alice ===============

Msg to Alice: First meg from Alice!

========= step 5.3: Bob send (First meg from Bob!) ===========

======== Whole sending message: ==============================

counter: 00000000

ciphertext:
eafa6ecc33c10332646e4a918e8eb3e56f44b272fe968e10ae5c3ae928bf72e504761d
52b94df2ebc5a923b7aafe38e4a41ec9fc87851a1b1bd85e5bfa93e59b

authtext:
94d602a4f42c1926e240851c6f1ea08a21caba75f1118840c3a82ebe56c18df3

========= step 5.4: Alice decrypte meg from Bob ==============

Msg to Bob: First meg from Bob!



======== All steps finish! ===================================
```