Week 11 Classwork due on Friday Nov 23, 22:00 Hour

Group 5

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

Exercise 1

Implement the final version of the key negotiation protocol. (You can use external libraries.)

Answers:

We implemented the final version of the key negotiation protocol by following the process outlined in week 11 lecture notes page 17.

The whole protocol contains 4 steps.

Step1:

Alice send Sa and N to Bob. We first set Sa=8 and Sb=10.

Step2:

Bob needs to generate the (g, p, q), B and $AUTH_{bob}$(also signature) where we set the signing message = g+p+q+B. Also, the g, p, q, B need to meet the requirements. (See in the choose_gpq() function in the codes).

Step3:

Alice first verify the signature and a number of conditions. After that, Alice generates K' and K, sending the A to Bob. Meanwhile, Alice generate the $AUTH_{Alice}$(also signature) and send to Bob where we set the signing message is A. (See in the receiveFromAlice() function in the codes).

Step4:

Finally, Bob first verify the signature and a number of conditions. After that, Bob generate K' and K (See in the bobReceive () function in the codes)

The result is shown in Figure 1. We observed that the Ks generated from Alice and Bob are the same. The code is written in Python 2.7

```
=========Alice sending Sa, N:==============
Sa is: 8
N is: 211914268459513145422251889616280500466027164339268494207107648957159941416 4
=========Bob sending (g,p,q),B,sig(Bob):==============
g is: 290
p is: 887
q is: 443
B is: 382
signature is:
26f13e501121e812148e6dafe724d57e66b05c473dedf24452b28004d4177e9f2be883e3bda9bac985368af5bb729bf3cf3a82125cc2e54213c4ffe44
1d0c5fec1a097ee53c782a11567a4410d24ec163e68b878c2d457a7a573a4e9ef7a2e95860a64573d2c75a4728c2a7c3ecd2cff7aca1db362e4866b81
b9bbcad9fb1e7b
=========Alice sending A,sig(Alice):============
A is: 81
K_prime is: 109
K: 36d1d1e2f4392d402e71d0e64f42de5d9bbdfe9ec999a7ead83ff1335767c235
signature is:
209dc2417b89c58992d2d8670c1c2213116af899d5cb07c80db908d0f07b4f2d9e94c5073c3d8128492e6cde6000a8e72d4a539ddc65b2a4343863aca
dd91162df1fe843cedda2ef6705133ed2a77cdc45fff8aa7e928a74df786e100bc4ce0cf958f23955e3bc9a1d32915412dbe51939a7d5db0478d45308
a1d1de32889e24
=========Bob verify and generate K============
K_prime_Bob is: 109
K_Bob is: 36d1d1e2f4392d402e71d0e64f42de5d9bbdfe9ec999a7ead83ff1335767c235
```

Figure 1

```
import random
from Crypto.Hash import SHA256
from Crypto.Signature import PKCS1_v1_5
from Crypto.PublicKey import RSA
import math
from Crypto.Signature import PKCS1_v1_5
from Crypto.PublicKey import RSA


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

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

def choose_gpq(Sa, Sb):
    alfa = 2
    g = 0
    s = max(Sa, Sb)
    if s > 2*Sb:
        return False
    while(True):
        p = generatePrime(s)
        if math.log(p,2) < s-1:
```

```
            continue
        q = (p-1)/2
        if not 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, alfa, publickey, signature

def receiveFromAlice(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 isPrime(q):
            print 'error'
        if not 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(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

#
Sa = 8
Sb = 10
N = random.randint(0, 2**256-1)
print '=========Alice sending Sa, N:=============='
print 'Sa is:', Sa
print 'N is:', N
print '=========Bob sending (g,p,q), B, signature(Bob):=============='
g, p, q, b, B, alfa, publickey, signature = choose_gpq(Sa, Sb)
print 'g is:', g
print 'p is:', p
print 'q is:', q
print 'b is:', b
print 'B is:', B
print 'signature is:', signature.encode('hex')
print '=========Alice sending A, signature(Alice):============='
a, A, K_prime, K, publickey_new, signature_new = receiveFromAlice(Sa, g, p, q, B, publickey,
signature)
print 'A is:', A
print 'K_prime is:', K_prime
print 'K:', K
print 'signature is:', signature_new.encode('hex')
print '=========Bob verify and generate K============'
K_prime_FromBob, K_FromBob = bobReceive(A, g, p, q, b, publickey_new, signature_new)
print 'K_prime_Bob is:', K_prime_FromBob
print 'K_Bob is:', K_FromBob
```

The resulting outputs are:

=========Alice sending Sa, N:==============

Sa is: 8

N is: 211914268459513145422251889616280500466027164339268494207107648957159941416 4

=========Bob sending (g,p,q),B,sig(Bob):==============

g is: 290

p is: 887

q is: 443

B is: 382

signature is: 26f13e501121e812148e6dafe724d57e66b05c473dedf24452b28004d4177e9f2be883e3bda9bac985368af5bb729bf3cf3a82125cc2e54213c4ffe441d0c5fec1a097ee53c782a11567a4410d24ec163e68b878c2d457a7a573a4e9ef7a2e95860a64573d2c75a4728c2a7c3ecd2cff7aca1db362e4866b81b9bbcad9fb1e7b

=========Alice sending A,sig(Alice):============

A is: 81

K_prime is: 109

K: 36d1d1e2f4392d402e71d0e64f42de5d9bbdfe9ec999a7ead83ff1335767c235

signature is: 209dc2417b89c58992d2d8670c1c2213116af899d5cb07c80db908d0f07b4f2d9e94c5073c3d8128492e6cde6000a8e72d4a539ddc65b2a4343863acadd91162df1fe843cedda2ef6705133ed2a77cdc45fff8aa7e928a74df786e100bc4ce0cf958f23955e3bc9a1d32915412dbe51939a7d5db0478d45308a1d1de32889e24

=========Bob verify and generate K============

K_prime_Bob is: 109

K_Bob is: 36d1d1e2f4392d402e71d0e64f42de5d9bbdfe9ec999a7ead83ff1335767c235

The parallel session launched should begin with the first step. You cannot skip to the 4th step directly. You may use the optimised mutual authentication protocol for the parallel session.

Exercise 2

Is the symmetric key based mutual authentication protocol (Slide 26) subject to reflection attack? If yes, describe the attack.

Answers:

A mutual authentication protocol requires each party to respond to a random challenge by the other party by encrypting it with a pre-shared key. Usually the same pre-shared key for communication is employed with a number of different users in this protocol. An attacker can easily compromise this protocol without using the correct key by employing a reflection attack on the protocol.

Reflection attacks make use of the mutual authentication process to trick a target user to reveal the secret key shared between him and the other user. During the mutual authentication process, a secret key is known to both the valid user and the server. In order to verify the shared secret key without transmitting it plainly over the network, they utilize a Diffie-Hellman-style scheme in which they each pick a nonce value, then request the hash of that value using the shared key. In a reflection attack, the attacker pretends to be the valid user and requests the hash of a random value from the server. When the server returns this value and requests its own value to be hashed, the attacker opens another connection to the server. This time, the hash value requested by the attacker is the value which the server requested in the first connection. When the server returns this hashed value, the attacker immediately uses it in the first connection, authenticating him successfully as the “valid” user (which in fact he is not).

In the class lecture, the mutual authentication uses five steps for authentication. It is susceptible to the reflection attack. The attacker first initiate a connection with Bob and when Bob send a nonce value for the attacker to hash, the attacker opens a second connection with Bob. In the second connection, the attacker will send the same nonce value to Bob to hash. When Bob returns with the hash value, the attacker immediately use it together with a nonce value in the first connection with Bob. Upon successful verification and returning the hash value, Bob will authenticate the attacker as the valid user (or Alice). The entire flow is shown in Figure 2. Therefore, the five step mutual authentication is equally susceptible to reflection attack as the optimized mutual authentication method.

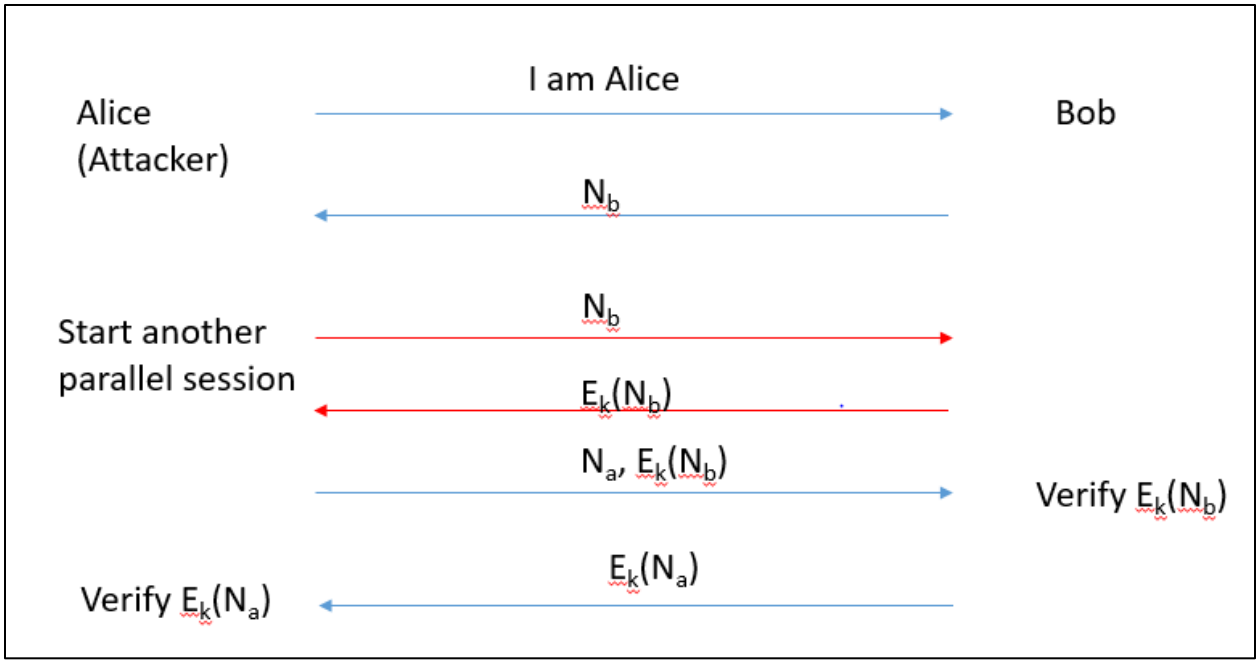
I am Alice
Alice
(Attacker)
Bob
N_b
N_b
Start another
parallel session
$E_k(N_b)$
N_a, $E_k(N_b)$
Verify $E_k(N_b)$
$E_k(N_a)$
Verify $E_k(N_a)$

Figure 2