HW2

Yi Xue 03/22/2016

1. (a) I can set the middle point between K1 and K2/K3, or K1/K2 and K3, can calculate the processed results on that point from both sides.

(b) One side requires 2^112 keys, the other side requires 2^56 keys.

(c) (2^112 + 2 ^56) * 64 bit , the dominant one is 2^112. 2^10 is approximately 10^3, 2^112 is about 10^33, 1TB is 10^12 bytes, 10^33 * 64 bit (8 Byte) = 8 * 10 ^21 TB , so total requires about 10^22 TB.

2. (a) Assume A already sent request to talk to B

(b) B can send a challenging nonce to A , and if A can get it correctly and confirm it to B, then A is validated.

B wanted to confirm A is on the other side, B would send a nonce , encrypted with A's public key, and send it out, after A got it, he would retrieve it, and pack it with other optional info, and encrypted them with B's public key , and sent back to B. After B got it, and he could validate it with the nonce he sent.

(c) (i) B sent a nonce to A, encrypted with A's public key
(ii) A got it, and extracted the nonce using the private key, then sent it back to B, encrypted with B's public key
(iii) B got the returned message, then decrypted and confirmed the nonce was correct, A was validated.

3.(a)
6 ^ 1 mod 11 = 6
6 ^ 2 mod 11 = 3
6 ^ 3 mod 11 = 7
6 ^ 4 mod 11 = 9
6 ^ 5 mod 11 = 10
6 ^ 6 mod 11 = 5
6 ^ 7 mod 11 = 8
6 ^ 8 mod 11 = 4
6 ^ 9 mod 11 = 2
6 ^ 10 mod 11 = 1
The remainders range from 1-10 and are non-repetitive. Thus 6 is the primitive root of 11.

(b) Y_A = a ^ X_A mod q
5 = 6 ^ X_A mod 11
from (a), X_A = 6

(c) Y_B = a ^ X_B mod q
4 = 6 ^ X_B mod 11
from (a), X_B = 8

(d) $K_{AB} = Y_B \wedge X_A \bmod q = 4 \wedge 6 \bmod 11 = 4$
(e) $K_{AB} = Y_A \wedge X_B \bmod q = 5 \wedge 8 \bmod 11 = 4$

4. (a)
$K = Y_A \wedge k \bmod q = 5 \wedge 2 \bmod 11 = 3$

$C1 = a \wedge k \bmod q = 7 \wedge 2 \bmod 11 = 5$
$C2 = KM \bmod q = 3 * 4 \bmod 11 = 1$

(b)
$K = C1 \wedge X_A \bmod q = 5 \wedge 2 \bmod 11 = 3$
$M = C2 * K^{-1} \bmod q = 3^{-1} \bmod 11 = 4$

5. (a) BG can replace the PU_B with his own public key PU_{BG} and send it to A.

(b) BG can replace the signed (B key || time) with the signed (BG key || time) to A.

(c) BG could include some outdated and compromised signed (B key || request) to A.

(d) BG can pretend he is B and create virtual circuit with A , because there is no authentication for B, which was provided by N1.

Programming problem:
(1) Code can perform two attacks, and returned the attacked results.
Code is written in Python, as Python has no limit on numbers, thus good for the application.

When numbers are small, log attack could be faster, when numbers are huge factor attack is far more efficient.

RSAattack.py

```
#!/usr/bin/python3

import math
import time



#the function will find the divisors, assume n is the product of two prime numbers
# the function itself does not validate n is the product of 2 prime numbers
def findPrimerDivisors(n) :

	#get the largest int of squre root of n
	sqrt = math.floor(math.sqrt(n))

	if n % 2 == 0 :
		return (2, n//2)

	for i in range(3, sqrt+1, 2) :
		if n % i == 0 :
			return (i, n//i)

	return (0, 0)



#using euclidean algorithm , return the coeffients tuple
#in tuple, first element is the remaining r,
# second the multiplicative coeffients q x -1
# a = q1 b + r1  =>  a + (-q1) b = r1
# b = q2 r1 + r2 =>  b + (-q2) r1 = r2
# ....
# the final coeff[1] is the inverse modulo of deKey

def inverseMod(a, b) :

	#basis case 1:  coprime
	if a % b == 1 :
		coeff = (1, a // b * (-1))
		return coeff
```

```
    #basis case 2: not coprime
        if a % b == 0 :
            print("a and b are not coprime, return 0,0")
            return (0, 0)

    #recursive cycles
        r = a % b
        q = a // b * (-1)

        coeff = inverseMod(b, r)
        newR = coeff[1]
        newQ = coeff[0] + coeff[1] * q
        return (newR, newQ)

#the output from findInverseModulo could be negative
#create the postive inverse modulo
def findDeKey(p, q, enKey) :
        totient = (p-1) * (q-1)
        deKey = inverseMod(totient, enKey)[1]
        if deKey < 0 :
            deKey += totient

        return deKey


def moduloPower(base, power, modulus) :
        if base == 0 :
            return 0

        answer = 1
        for i in range (0, power) :
            answer = answer * base % modulus
        return answer


def factorAttack(e, n, c) :
        start = time.clock()
        (p,q) = findPrimerDivisors(n)
        deKey = findDeKey(p, q ,e)
        m = moduloPower(c, deKey, n)
        stop = time.clock()
        print("Factor attack output: ")
        if deKey == 0 :
            print ("Attack failed")
        else:
            print("p =", p)
            print("q =", q)
            print("d =", deKey)
```

```
                print("M =", m)
                print("Used time in miliseconds : ", (stop - start) * 1000, "\n")


def discreteLogAttack(e, n, c) :
        start = time.clock()
    # m is intialized to -1, in case not found, -1 is indicative
        m = -1
        if c == 0:
                m = 0
        elif c == 1:
                m = 1
        else:
                for i in range(2, n) :
                        if moduloPower(i, e, n) == c :
                                m = i
                                break
        stop = time.clock()
        print("Discrete log attack : ")
        if m == -1 :
                print ("Attack failed")
        else :
                print ("M = ", m)
                print("Used time in miliseconds : ", (stop - start) * 1000, "\n")


print("RSA attack demo:")

while True:

        userInput = input("Please input public key, n, and ciphertext, separeted by space : ")
        e = int(userInput.split(" ") [0])
        n = int(userInput.split(" ") [1])
        c = int(userInput.split(" ") [2])

        factorAttack(e, n, c)
        discreteLogAttack(e, n, c)

        quit = input("Do you want to quit ? y for quit, all other keys for continue :  ")
        if (quit == "Y" or quit == "y") :
                break
```

Output:

```
yi@yi-Inspiron-5437:~/codes/python/crypto$ ./RSAattack.py
RSA attack demo:
Please input public key, n, and ciphertext, separeted by space : 7 15 4
Factor attack output:
p = 3
q = 5
d = 7
M = 4
Used time in miliseconds :  0.04200000000000037

Discrete log attack :
M =  4
Used time in miliseconds :  0.009999999999999593

Do you want to quit ? y for quit, all other keys for continue :
Please input public key, n, and ciphertext, separeted by space : 13 527 289
Factor attack output:
p = 17
q = 31
d = 37
M = 51
Used time in miliseconds :  0.04300000000000137

Discrete log attack :
M =  51
Used time in miliseconds :  0.13200000000000017

Do you want to quit ? y for quit, all other keys for continue :
Please input public key, n, and ciphertext, separeted by space : 2003 2511739
1978837
Factor attack output:
p = 1249
q = 2011
d = 1202267
M = 950
Used time in miliseconds :  231.431

Discrete log attack :
M =  950
Used time in miliseconds :  354.43699999999995

Do you want to quit ? y for quit, all other keys for continue :  y
```