Министерство науки и высшего образования Российской Федерации
Санкт-Петербургский Политехнический Университет Петра Великого
—
Институт компьютерных наук и кибербезопасности
Высшая школа кибербезопасности

ЛАБОРАТОРНАЯ РАБОТА № 2

«Атаки на криптосистему RSA, использующие данные о показателях»

по дисциплине «Криптографические методы защиты безопасности»

Выполнил
студент гр. 5151004/90101 Кондачков Е.Д.

Преподаватель
Ассистент Ярмак А.В.

Санкт-Петербург
2023

Содержание

1. Цель работы

Изучение уязвимостей криптосистемы RSA по отношению к атакам, использующим показатели, обладающие определенными свойствами, генерация параметров, криптосистемы RSA.

2. Задачи работы

1. Изучить описание трех атак на криптосистему RSA, приведенное в методическом пособии к лабораторной работе;

2. Реализовать алгоритм разложения составного числа на множители по известным показателям RSA;

3. Реализовать атаку Винера на криптосистему RSA;

4. Реализовать бесключевое дешифрование сообщения в случае малого порядка элемента e в группе $(\mathbb{Z}/\varphi(n)\mathbb{Z})^*$.

3. Ход работы

1.1 Реализация разложения составного числа на множители по известным показателям RSA

В качестве первой атаки рассматривалось разложения составного числа на множители по известным показателям RSA. Основной задачей данной атаки являлось нахождение разложения числа n на множители p и q, что обычно является весьма трудоемкой задачей. Алгоритм основывается на том, что отправитель и получатель имеют один и тот же модуль n. Это означает, что, например, отправитель, зная собственные открытый показатель e и закрытый показатель d, а также открытый показатель e' получателя и общий модуль n может найти числа p и q – делители общего модуля. Уже по известным p и q можно было найти d получателя.

Алгоритм данной атаки выглядит следующим образом:

Вход. Число n, показатели e_B, d_B такие, что $e_B d_B \equiv 1 \, (mod\ \varphi(n))$, открытый показатель e_A другого пользователя.

Выход. Делители p и q числа n, закрытый показатель d_A другого пользователя.

1. Представить разность $e_B\ d_B\ -\ 1$ в виде $2^f s$, где s – нечетное число.

2. Выбрать случайное $a\ \in\ \mathbb{Z}\ /\ n\mathbb{Z}$ и вычислить $b\ \leftarrow\ a^s\ (mod\ n)$.

3. Вычислять $b^{2^0}\ \equiv\ b\ (mod\ n)$, $b^{2^1}\ \equiv\ \left(b^{2^0}\right)^2\ (mod\ n)$, $b^{2^2}\ \equiv\ \left(b^{2^1}\right)^2\ (mod\ n)$, ... до получения l такого, что $b^{2^l}\ \equiv\ 1\ (mod\ n)$. Если $b^{2^{l-1}}\ \equiv\ -1\ (mod\ n)$, то вернуться на шаг 2, иначе положить $t\ \leftarrow\ b^{2^{l-1}}\ (mod\ n)$.

4. Положить $p \leftarrow$ НОД$(t\ +\ 1, n)$, $q\ \leftarrow$ НОД$(t\ -\ 1, n)$.

5. Вычислить $\varphi(n)\ =\ (p\ -\ 1)(q\ -\ 1)$ и найти закрытый ключ другого пользователя: $d_A\ \equiv\ e_A\ -1\ (mod\ \varphi(n))$.

6. Результат: (p, q), d_A.

Реализация данного алгоритма в коде представлена на рисунке 1.

```python
def factorization_attack(e_a, e_b, d_b, n):

    s = e_b * d_b - 1
    while (s % 2 == 0):
        s = s // 2

    while True:
        t = n-1
        a = rnd.randrange(1, n)
        b = pow(a, s, n)
        x = b
        y = 0
        while True:
            y = x #b^2^(l-1)
            x = pow(x, 2, n)#b^2^l
            if x == 1:
                if y != -1:
                    t = y
                break
        if t != n-1 and t != 1:
            break

    p, q = GCD(t + 1, n), GCD(t - 1, n)
    phi = (p-1)*(q-1)

    d_a = inverse(e_a, phi)

    return p, q, d_a
```

Рисунок 1 – Реализация атака разложения на множители

1.2 Реализация атаки Винера на криптосистему RSA

Второй атакой была атака Винера, позволяющая получить закрытый показатель d, зная открытый показатель e и модуль n, то есть в случае получения открытого ключа пользователя. Данная атака реализуется при помощи разложения отношения $\frac{e}{n}$ в виде непрерывной дроби и нахождения подходящей к ней дроби $\frac{P_i}{Q_i}$. При нахождении корректной дроби, число в ее знаменателе будет совпадать с d, обратным уже известному e. Важно отметить, что из-за сложности и некоторой непредсказуемости описанного механизма параметры RSA для данной атаки были зафиксированы.

Алгоритм выглядит следующим образом:

Вход. Число n, показатель e.

Выход. Закрытый показатель d.

1. Представить $\frac{e}{n}$ в виде обыкновенной непрерывной дроби $[0;\ a_1\ , a_2\ , \ldots , a_l\ ]$, где $l\ \approx \log_2 n$.

2. Для $i\ =\ 1, \ldots , l$ выполнить следующие действия.

2.1. Вычислить i-ю подходящую дробь $\frac{P_i}{Q_i}$ к $\frac{e}{n}$: $\frac{P_i}{Q_i} = \frac{a_i P_{i-1} + P_{i-2}}{a_i Q_{i-1} + Q_{i-2}}$, где $P_{-1} = 1, Q_{-1} = 0, P_0 = 0, Q_0 = 1$.

2.2. Для произвольного сообщения $m \in \mathbb{Z} / n\mathbb{Z}$ проверить выполнение сравнения $(m^e)^{Q_i} \equiv m (mod\ n)$. Если сравнение выполняется, то положить $d \leftarrow Q_i$ и перейти на шаг 3.

3. Результат: d.

Реализация данной атаки представлена на рисунке 2. Параметры, выбранные для демонстрации атаки представлены на рисунке 3.

```
def wiener_attack(e, n):
    a = get_fraction(e, n)
    l = len(a)
    q = [0, 1, 0]
    m = rnd.randint(1, n-1)
    d = -1
    for i in range (1, l):
        q[2] = a[i] * q[1] + q[0]
        if pow(m, e*q[2], n) == m:
            d = q[2]
        q[0], q[1] = q[1], q[2]
    return d
```

Рисунок 2 – Реализация атаки Винера

```
        n = 15912080205244042782156159879724579419648676200728221361489953862529894O
    n = 15912080205244042782156159879724579419648676200728221361489953862529890765077907650779131236693265184430577558097325112614249221037779381730065276289572657846054735951416019146072056202596944863828596839039641936885292147084169738947447455523984819409275749168293471541000773697370089458024142902661248010563034292830
        e = 11344248856885807164295503665454231487754375121089537309789856526927258776275114297414196034298992736180410481299229993278201451840436370855751788204922795778488609359900203381710724197371026038017982542547796382021400514829497189707464973191055639707115355334522474997824622403851742433216683635618563112299
        d = 2340489565062645055447388848706575650805728883985056825422972181184277888053
9
```

Рисунок 3 – Параметры RSA для демонстрации атаки Винера

1.3 Реализация бесключевого дешифрования сообщения

В качестве последней атаки был рассмотрен метод дешифрования сообщений без использования закрытого показателя d, что реализуемо, если порядок открытого показателя e мал. Для демонстрации этой атаки размерность всех параметров RSA была сильно снижена в угоду скорости работы программы.

Алгоритм данной атаки таков:

Вход. Составное число n, шифртекст c, открытый показатель e.

Выход. Открытый текст m.

1. Для $i = 1, 2, \ldots$ выполнить следующие действия.

1.1. Вычислить $c_i \equiv c_{i-1}\text{^}e \equiv c_0^{e^i} \pmod n$, где $c_0 \leftarrow c$.

1.2. Проверить выполнение сравнения $c_i \equiv c \pmod n$. Если сравнение выполняется, то положить $m \leftarrow c_{i-1} \equiv c_0^{e^{ord(e)-1}} \pmod n$ и перейти на шаг 2.

2. Результат: m.

Генерация параметров представлена на рисунке 4. Реализация алгоритма представлена на рисунке 5.

```
p = getPrime(32, randfunc=get_random_bytes)
q = getPrime(32, randfunc=get_random_bytes)
e = 0
#init base parameters
n = p * q
v = (p-1)*(q-1)
while GCD(e, v) > 1:
    e = getPrime(32, randfunc=get_random_bytes)
d = inverse(e, v)
```

Рисунок 4 – Генерация параметров для демонстрации атаки

```
def keyless_decryption_attack(c, e, n):
    c_vector = [c, 0]
    for i in range(n):
        c_vector[1] = pow(c_vector[0], e, n)
        print(c_vector[0])
        if c_vector[1] % n == c:
            m = c_vector[0]
            break
        c_vector[0] = c_vector[1]
    return m
```

Рисунок 5 – Реализация атаки

1.4 Генерация безопасных параметров RSA

Для программы были выбраны большие простые числа p и q с разницей в 12 байт. Также они являются безопасными, то есть $p = 2p_1 + 1$, где p_1 – простое число. Показатель e был выбран 16 бит, не слишком большой или маленький.

Программа принимает на выход битовую длину модуля RSA, на выходе возвращает параметры криптосистемы: n, e, d. Результат отражен на рисунке 6.

```
Generated parameters:
**************************************************n = 27143685659020457775755043855
95364696113050767711445364843814324573824305097385184377924228535300384954187098164
82197350305923568911484688679421945376911191408331434435134528607956168085087589511
74037565119175865141286855549391874814050956383165475633878050365869233757770102755
74082133396800294737362396491260 29
**************************************************e = 63689
**************************************************d = 13750630000512428357753469750
95367277793519602591374856746389884721849414807298522331475565787858682349572022408
73331584893314670152775863894124097499421591837508582768044185824074987533830764505
22325020031486401608499524391528096658169039261375578422222714208788707721042551650
58345937221909591717774633539735 93
```

Рисунок 6 – Результат работы программы генерации параметров

Сгенерированные параметры были проверены утилитой Cryptools.

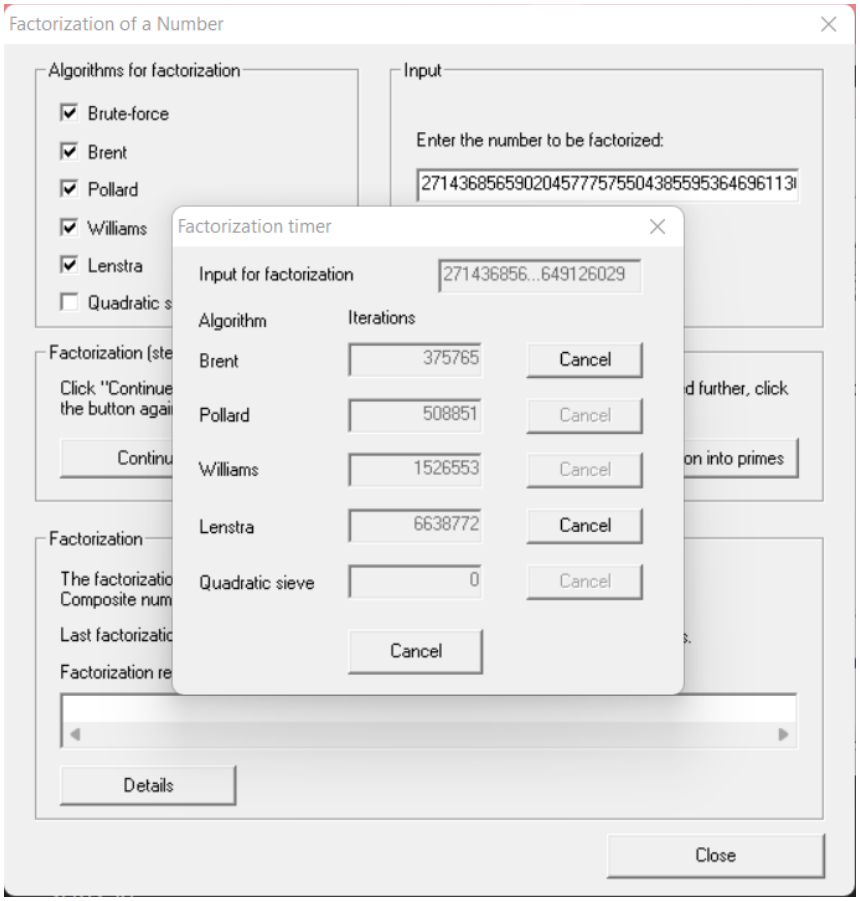

Рисунок 7 – Разложение модуля на множители

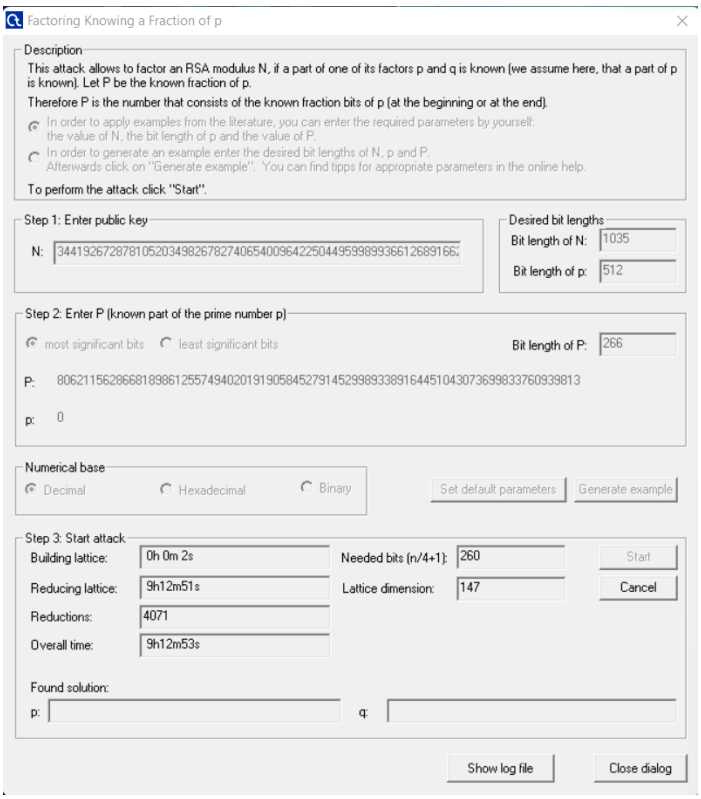

Рисунок 8 – Нахождение p и q по части p

4. Ответы на контрольные вопросы

1. При каком условии возможна атака Винера?

Атака Винера возможна, если не выполняется условие $d > \frac{1}{3}\sqrt[4]{n}$

2. Перечислите методы противодействия атаке Винера.

Соблюдение условия $d > \frac{1}{3}\sqrt[4]{n}$.

3. Почему числа p и q необходимо выбирать так, чтобы $p-1, q-1$ имели большие простые делители?

Для усложнения поиска числа n при помощи алгоритмов факторизации.

4. Предложите способы защиты от атаки бесключевого дешифрования широковещательных сообщений без отказа от использования общих малых закрытых показателей.

После превышения количества блоков шифртекстов обновлять параметры криптосистемы, чтобы сменить параметр e. Не допускать превышения количества блоков шифртекстов над параметром e путем увеличения размера блока шифртекста

5. Выводы

В рамках выполнения настоящей лабораторной работы были изучены методы реализации атак на криптосистему RSA. Для реализации атак на криптосистему RSA при генерации ее параметров нужно допустить ошибки, которые были описаны и изучены в методических материалах к данной лабораторной работе. По итогу проведения различных атак на криптосистему RSA генератор параметров криптосистемы был оснащен дополнительными проверками для повышения уровня криптостойкости криптосистемы RSA.

Приложение А

Листинг разработанной программы на языке Python3:

```
import random as rnd
import math
import time
from sympy import isprime, sqrt
from Crypto.Util.number import getPrime, inverse, GCD
from Crypto.Random import get_random_bytes

#--------FACTORIZATION BLOCK--------#

def factorization_attack(e_a, e_b, d_b, n):

    s = e_b * d_b - 1
    while (s % 2 == 0):
        s = s // 2

    while True:
        t = n-1
        a = rnd.randrange(1, n)
        b = pow(a, s, n)
        x = b
        y = 0
        while True:
            y = x #b^2^(l-1)
            x = pow(x, 2, n)#b^2^l
            if x == 1:
                if y != -1:
                    t = y
                break
        if t != n-1 and t != 1:
            break

    p, q = GCD(t + 1, n), GCD(t - 1, n)
    phi = (p-1)*(q-1)

    d_a = inverse(e_a, phi)
```

```
            return p, q, d_a

        def get_e_and_d(phi):
            e = 0
            while GCD(e, phi) > 1:
                e = getPrime(1024, randfunc=get_random_bytes)
            d = inverse(e, phi)
            return e, d

        def get_first_program_param():
            p = getPrime(1024, randfunc=get_random_bytes)
            q = getPrime(1024, randfunc=get_random_bytes)
            e_a = 0
            e_b = 0
            n = p * q
            phi = (p-1)*(q-1)
            e_a, d_a = get_e_and_d(phi)

            while True:
                e_b, d_b = get_e_and_d(phi)
                if e_a != e_b and d_a != d_b:
                    break
            return e_a, e_b, d_a, d_b, n

        def first_program():
            e_a, e_b, d_a, d_b, n = get_first_program_param()

            print(f' n = {n}\n\
                  e_a = {e_a}\n\
                  d_a = {d_a}\n\
                  e_b = {e_b}\n\
                  d_b = {d_b}')

            p_predicted, q_predicted, d_a_predicted = factorization_attack(e_a, e_b,
d_b, n)
            print(f'\
                  Predicted:\n\
                  p by factorisation {p_predicted}\n\
                  q by factorisation {q_predicted}\n\
```

```
            d_a by factorisation {d_a_predicted}')

    if d_a_predicted == d_a:
        print(f'Success')
    else:
        print(f'Failure')

#--------WIENER BLOCK--------#

def get_fraction(e, n):
      a, q = divmod(e, n)
      t = n
      x = [a]

      while q != 0:
            next_t = q
            a, q = divmod(t, q)
            t = next_t
            x.append(a)

      return x

def wiener_attack(e, n):
    a = get_fraction(e, n)
    l = len(a)
    q = [0, 1, 0]
    m = rnd.randint(1, n-1)
    d = -1
    for i in range (1, l):
        q[2] = a[i] * q[1] + q[0]
        if pow(m, e*q[2], n) == m:
            d = q[2]
        q[0], q[1] = q[1], q[2]
    return d

def second_program():

    n                                                                    =
15912080205244042782156159879724579419648676200728221361489953862529894076507791
```

```
123669326518443057755809732511261424922103777938173006527628957265784605473595141
601914607205620259694486382859683903964193688529214708416973894744745552398481940
927574916829347154100077369737008945802414290266124801056303429283
          e                                                                                =
113442488568858071642955036654542314877543751210895373097898565269272587762751142
974141960342989927361804104812992299932782014518404363708557517882049227957784886
093599002033817107241973710260380179825425477963820214005148294971897074649731910
556397071153553345224749978246224038517424332166836356185631122 99
          d                                                                                =
234048956506264505544738884870657565080572888398505682542297218118427788805 39

          print(f'\
              n = {n}\n\
              e = {e}\n\
              d = {d}')

          d_predicted = wiener_attack(e, n)
          print(f'Predicted d by wiener {d_predicted}')

          if d == d_predicted:
              print(f'Success')
          else:
              print(f'Failure')

      #--------KEYLESS DECRYPTION BLOCK--------#

      def keyless_decryption_attack(c, e, n):
          c_vector = [c, 0]
          for i in range(n):
              c_vector[1] = pow(c_vector[0], e, n)
              print(c_vector[0])
              if c_vector[1] % n == c:
                  m = c_vector[0]
                  break
              c_vector[0] = c_vector[1]
          return m

      def third_program():
          m = 156
```

```
    p = getPrime(32, randfunc=get_random_bytes)
    q = getPrime(32, randfunc=get_random_bytes)
    e = 0
    #init base parameters
    n = p * q
    v = (p-1)*(q-1)
    while GCD(e, v) > 1:
        e = getPrime(32, randfunc=get_random_bytes)
    d = inverse(e, v)

    c = pow(m, e, n)

    print(f'\
        n = {n}\n\
        e = {e}\n\
        d = {d}\n\
        m = {m}\n\
        c = {c}')
    start = time.time()
    m_predicted = keyless_decryption_attack(c, e, n)
    end = time.time() - start
    print(f'Predicted d by wiener {m_predicted}')
    print(f'spended time = {end} seconds')
    if m == m_predicted:
        print(f'Success')
    else:
        print(f'Failure')

#--------PARAMETERS GENERATOR-------#

def gen_security_parameters():
      # p и q - безопасные простые числа, z = 2*z1 + 1
      while True:
            p = getPrime(512, randfunc=get_random_bytes)
            p_1 = (p - 1) // 2
            if(isprime(p_1) == True):
                  break
      while True:
            q = getPrime(512 + 12, randfunc=get_random_bytes) # разница в
```

```
12 байт между p и q
                q_1 = (q - 1) // 2

                if(isprime(q_1) == True):
                      break

          n = p * q
          v = (p - 1) * (q - 1)

          while True:
                e = 2
                while (GCD(e, v) > 1):
                      e = getPrime(16, randfunc=get_random_bytes) # 16 байт, не
слишком большое и не слишком маленькое
                d = inverse(e, v)
                if(d >= sqrt(sqrt(n))):
                      break

          return n, e, d, p, q


      def main():

          while True:
              print('''
      Warning: You will not be able to decrypt the file or verify
      the signature if the action was  not performed at the current start of the
program.

      Available fighters:
      1. Harry Potter (factorization)
      2. Tom Sawyer (wiener)
      3. Optimus Prime (keyless decryption)
      4. John Snow (generate parameters)
      ''' )
              command = input("Choose your fighter: ")

              if command == '1' or command == 'Harry Potter':
                  first_program()
```

```
        elif command == '2' or command == 'Tom Sawyer':
            second_program()
        elif command == '3' or command == 'Optimus Prime':
           third_program()
        elif command == '4' or command == 'John Snow':
            n, e, d, p, q = gen_security_parameters()
            print(f'Generated parameters:\n\
{"*"*50}\nn = {n}\n\
{"*"*50}\ne = {e}\n\
{"*"*50}\nd = {d}\n\
{"*"*50}\np = {p}\n\
{"*"*50}\nq = {q}')
        else:
            print("incorrect input, try again")

if __name__ == '__main__':
    main()
```