Q1 Team Name

0 Points

Enciphered

Q2 Commands

10 Points

List the commands used in the game to reach the ciphertext.

==> exit1
==> exit3
==> exit4
==> exit4
==> exit1
==> exit3
==> exit4
==> exit1
==> exit3
==> exit2
==> read

Q3 Analysis

60 Points

Give a detailed description of the cryptanalysis used to figure out the password. (Use Latex wherever required. If your solution is not readable, you will lose marks. If necessary the file upload option in this question must be used TO SHARE IMAGES ONLY.)

The discussion is start with an explanation of the commands used in the game to reach the ciphertext. The screen already suggested that the right sequence of comments can be found by a brute force search of possible comments like exit1, exit2, exit3 and exit4. Here we have successfully encountered the correct list of commands. Whenever we got a new window, we observed that there is a new sequence of hexadecimal values.

Sequentially listed all such commands including hexadecimal values.

---

exit1: 59 6f 75 20 73 65 65
exit3: 20 61 20 47 6f 6c 64
exit4: 2d 42 75 67 20 69 6e
exit4: 20 6f 6e 65 20 63 6f
exit1: 72 6e 65 72 2e 20 49
exit3: 74 20 69 73 20 74 68
exit4: 65 20 6b 65 79 20 74
exit1: 6f 20 61 20 74 72 65
exit 3: 61 73 75 72 65 20 66
exit2: 6f 75 6e 64 20 62 79

At the end hexadecimal string is " 59 6f 75 20 73 65 65 20 61 20 47 6f 6c 64 2d 42 75 67 20 69 6e 20 6f 6e 65 20 63 6f 72 6e 65 72 2e 20 49 74 20 69 73 20 74 68 65 20 6b 65 79 20 74 6f 20 61 20 74 72 65 61 73 75 72 65 20 66 6f 75 6e 64 20 62 79". As a cryptographer, we need to find the significance of this string, so converted to binary string and then ASCII. Fortunately it produces a meaningful text: "You see a Gold-Bug in one corner. It is the key to a treasure found by" (say it as "guess" plaintext 1). In addition, we have another meaningful text in the final window (obtained after entering read). This is :-

"Enciphered: This door has RSA encryption with exponent 5 and the password is " (say it as "guess"plaintext 2)

From this screen we got the information about some RSA parameters. The screen mentioned about the public key value N, e, and ciphertext C.

$$N = 84364443735725034864402554533826279174703893439$$

Public key exponent $e = 5$ and ciphertext

$$C = 29690234374361835123368700863265495818333700503$$

But there is no information about private key exponent d and message M. Another excellent observation is about the value of the public key exponent e. From literature and class notes it is known to us that without integer factoring one can extract (knowing some bits of the message) the message. This attack is termed as "Breaking Low Exponent RSA". This most powerful attacks on low public exponent RSA are based on a theorem due to Coppersmith.

Let us detailed about the term low exponent attack. In textbook RSA, generally user is always try to reduce or optimize encryption or signature-verification time by choosing a small public exponent e. The smallest possible value for e is 3, but to defeat certain attacks the value $e = 2^{16} + 1 = 65537$ is recommended in the international and national standard. When the value $2^{16} + 1$ is used, signature verification requires 17 multiplications, as opposed to roughly 1000 when a random $e \leq \phi(N)$ is used. Let us state the Coppersmith algorithm, for more details one may go thorough the listed references related tow Coppersmith algorithm.

Theorem: (Coppersmith) Suppose N be an integer and $f \in \mathbb{Z}[x]$ be a monic polynomial of degree d. Set $X = N^{\frac{1}{d} - \epsilon}$ for some $\epsilon > 0$. Then, given $\langle N, f \rangle$ attacker can efficiently compute all integers $|x_0| < X$ satisfying $f(x_0) = 0 \mod N$. The time complexity of the method is dominated by the time it takes to run the famous LLL algorithm on a lattice of dimension $O(w)$ with $w = \min(\frac{1}{\epsilon}, \log_2 N)$.

Let us plot this algorithm in text book RSA and analyze a bit more. However we will not detailed about the main Coppersmith algorithm, since Professor includes these machinery
in video lectures.

As we know that the encrypted message is $C = M^e \mod n$. Suppose that the first l bits of the message M is a secret string x, and the remaining bits of m form a header b that is known to all. In other words, $M = b \cdot \cdot 2^l + x$, where b is known and x in unknown. Since, $(b \cdot \cdot 2^l + x)^e = C \mod N$, we get an equation $g(x) = x^e + \sum_{i=0}^{e-1} a_i x^i = 0 \mod N$, where $a_0, \cdots, a_{e-1}$ are known and are all less than N. Decryption of C is equivalent to finding a root of $g \mod N$.

We intend to solve this modular root finding problem using lattice basis reduction. The idea is to transform the modular root finding problem into a root finding problem over integers. The latter problem can then be solved using the LLL algorithm.

So we believe that , it is known about few bits of the original message. Goal is to retrieve the entire plaintext. We hope that the unknown bits may help us to clear the level six of the game. So far, we mentioned about two tentative known message bit string. Both strings are used to check whether Coppersmith algorithm provides a solution or not.

Let re-write the mathematical equation behind this

$$f(x) = (P + x)^e \mod N$$

One thing is that we are curious about the bit size of the solution x_0. Basically this question is answered by the Coppersmith theorem . Actually this theorem come up with an efficient algorithm to find all roots of $f \mod N$ that are less than $X = N^{\frac{1}{d}}$. This ensures that x_0 the roots of the polynomial has degree at most d.

Now use the binary encoding of "guess" hexadecimal string. Our implementation can be split into two parts, one part is basic preprocessing for Coppersmith algorithm and the main function for Coppersmith algorithm. The preprocessing part defines and deduces all useful parameters needed for the algorithm. For the implementation of Coppersmith algorithm we took a help from a good github source. They have implemented the Coppersmith algorithm, "Coppersmith_hograve_univariate(pol, N, beta, mm, tt, XX)" is the function to handle the algorithm.

As mentioned we are avoiding to put mathematical details of the programming details. Here"pol" is a polynomial and it is defined through SAGEMATH libary. The github source already suggested a set of default values for the parameters.

```
dd = f.degree()
beta = 1
epsilon = beta / 7
mm = ceil(beta**2 / (dd * epsilon))
```

tt = floor(dd * mm * ((1/beta) - 1))
XX = ceil(N**((beta**2/dd) - epsilon))
roots = coppersmith_howgrave_univariate(f, N, beta, mm, tt, XX)

Now we deploy the known bits of the entire message to retrieve the entire message. After running "Coppersmith_hograve_univariate" with guess plaintext1 we get no solution, however running with guess plaintext2 algorithm outputs a root. Thus correct value for P is binary encoding of "Enciphered: This door has RSA encryption with exponent 5 and the binary password is

$$010000110011100001011001010100000011011101101111$$

Lets break it into eight bit chunks, that is "01000011" "00111000" "01011001" "01010000" "00110111" "01101111" "01001100" "01101111" "00110110" "01011001". Now convert each chunk to ASCII character.

----------BINARY--TO--ASCII----------

01011001 ==> Y 00110110 ==> 6 01101111 ==> o
01001100 ==> L 01101111 ==> o 00110111 ==> 7
01010000==> L 01011001 ==> Y 00111000 ==>8
01000011 ==>C

----------BINARY--TO--ASCII----------

Collect all the characters and get "C8YP7oLo6Y". We enter "C8YP7oLo6Y" instead of "read" command and successfully able to clear the level.

References:

1. Twenty Years of Attacks on the RSA Cryptosystem by Dan Boneh
2. Don Coppersmith. Finding small solutions to small degree polynomials.
3. Prof. Chandan Saha's class notes https://www.csa.iisc.ac.in/~chandan/courses/CNT/notes/lec17.pdf
4. Prof. Oded Regev's class notes https://cims.nyu.edu/~regev/teaching/lattices_fall_2004/ln/rsa.pdf

5. The LLL Algorithm Survey and Applications by Phong Q. Nguyen and Brigitte Vallée.
Link: https://link.springer.com/book/10.1007/978-3-642-02295-1
6. Prof. Manindra Agrawal's class notes.
7. https://github.com/mimoo/RSA-and-LLL-attacks#readme
8. Factoring polynomials with rational coefficients by A. K. Lenstra, H. W. Lenstra Jr. & L. Lovász.
9. Binary to ASCII on geeksforgeeks

No files uploaded

Q4 Password

10 Points

What was the final command used to clear this level?

C8YP7oLo6Y

Q5 Codes

0 Points

It is MANDATORY that you upload the codes used in the cryptanalysis. If you fail to do so, you will be given 0 for the entire assignment.

Enciphered code.ipynb Download

Note for user: Please make sure that you installed sage-math v9.0 or above (preferable v9.3) in your system before running this code.

In [14]:
```
import numpy as np
import math
```

In [15]:
```
def calculate_mm(pol):
    return ceil(1**2 / (pol.degree() * (1/7)))
```

In [16]:
```
def calculate_tt(x, pol, beta):
    return floor(pol.degree() * x *
((1/beta) - 1))
```

In [17]:
```
def calculate_XX(N, pol):
    return ceil(N**((1**2/pol.degree()) -
(1/7)))
```

In [18]:
```
def get_unicode(p):
    hex_p = ['{0:08b}'.format(ord(x)) for
x in p ]
    bin_p = ''.join(hex_p)
    return bin_p
```

In [19]:
```
def get_epsilon(x):
    ans = x/7
    return ans
```

In [20]:
```
def get_Polynomial(fx, M, e, C):
    ans = ( fx + M)^e - C
    return ans
```

In [21]:
```
def get_key_in_binary(x):
    ans = bin(x[0])[2:]
    return ans
```

Defining RSA Algorithm for final analysis

In [23]:
```
def run_RSA_Algo(p, len_M,e,C,N):
    label, beta = 0, 1
    eps = get_epsilon(beta) ## Get
epsilon from beta i.e. 1
    unicode_p = get_unicode(p)  ## Get
unicode form
    for i in range(0, len_M+1, 4):
        fx = ( int(unicode_p, 2 )*(2**i)
) # fx is the monic polynomial f(x) = (p
+ m)^e - c whose roots we have to find
        P.<M> = PolynomialRing(Zmod(N))
        pol = get_Polynomial(fx, M, e, C)
        XX = calculate_XX(N, pol)
        mm = calculate_mm(pol)
        tt = calculate_tt(mm, pol, beta)

        roots =
Coppersmith_hograve_univariate(pol, N,
beta, mm, tt, XX)
        if roots:
            label = 1


print(get_key_in_binary(roots))
```

```
            break

    if label==0: ## in this case, no
roots found
        print('NO SOLUTION. PLZ TRY
AGAIN')
```

Defining Copper Hograve Univariate algorithm

In [22]:

```
def
Coppersmith_hograve_univariate(polynomial,
modulus, beta, m, t, X):
    polynomials = [] # Polynomials
    polynomial =
polynomial.change_ring(ZZ)
    x =
polynomial.change_ring(ZZ).parent().gen()
    degree = polynomial.degree()
    for i in range(m):
        for j in range(degree):
            polynomials+=[(x * X)**j *
modulus**(m - i) * polynomial(x * X)**i ]
    for i in range(t):
        polynomials+=[(x * X)**i *
polynomial(x * X)**m]
    y = degree * m + t
    lattice_B = Matrix(ZZ, y)
    for i in range(y):
        for j in range(i+1):
            lattice_B[i, j] =
polynomials[i][j]
    lattice_B = lattice_B.LLL()
    polynomial = 0
    for i in range(y):
        polynomial += x**i * lattice_B[0,
i] / X**i

    possible_roots = polynomial.roots()

    roots = []
    for root in possible_roots:
        if root[0].is_integer():
            result =
polynomial(ZZ(root[0]))
            if gcd(modulus, result) >=
modulus^beta:
                roots+=[ZZ(root[0])]

    return roots
```

Initializing parameters

In [24]:

```
p = "Enciphered: This door has RSA encryption w
exponent 5 and the password is "
leng = 200
e = 5 # exponent = 5 given in ssh shell
c =
29690234374361835123368700863265495818333700503
```

```
n =
84364443735725034864402554533826279174703893439
```

Run RSA Algorithm

NOTE: : copy the answer printed in this cell to the next one as per the instruction given there

In [25]:

```
run_RSA_Algo(p, leng, e,c,n)
```

```
1000011001110000101100101010000001101110110111
```

Convert from binary chunks of size 8 bits each into ASCII

Note: copy the root printed above and pad it with a '0' bit in beginning manually and assign it to binary_int variable

In [26]:

```
binary_int =
int("0100001100111000010110010101000000110111010
 2);
byte_number = binary_int.bit_length() + 7 // 8
binary_array = binary_int.to_bytes(byte_number,
ascii_text = binary_array.decode()
print(ascii_text)
```

In [ ]:

GROUP

Anindya Ganguly
Gargi Sarkar
Utkarsh Srivastava

View or edit group

TOTAL POINTS

80 / 80 pts

QUESTION 1

Team Name — **0** / 0 pts

QUESTION 2

Commands — **10** / 10 pts

QUESTION 3

Analysis — **60** / 60 pts

QUESTION 4

Password — **10** / 10 pts

QUESTION 5

Codes — **0** / 0 pts