Q1 Team Name
0 Points

Turing

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.)

After logging into the server and entering level 6 we are in a chamber with several exits numbered 1 to 5 and only exit5 is closed. Then, we entered one of the gates with the “exit1” command and a strange number appeared(we wonder what that might be). Also, it was given that there were 4 doors open and exit 5 is closed. We tried entering gates randomly and we noticed that in a given chamber with a number, particular exit command is taking us to a particular chamber. So, we thought the number might be some kind of id for a room, and the chambers were connected to different chambers through these gates. So, taking the numbers as id, we made the whole map of all the chambers with all the paths, and we got a total of 10 chambers interconnected. We were checking the read command in all the chambers, and in the chamber with id “6f 75 6e 64 20 62 79”, we found the cryptoanalysis problem.

n=84364443735725034864402554533826279174703893439763343343863260342756678609216895093779263028809246505955647572176682669445270008816481771701417554768871285020442403001649254405058303439906229201909599348669565697534331652019516409514800265887388539283381053937433496994442146419682027649079704982600857517093

Turing: This door has RSA encryption with exponent 5 and the password is 49405883664112596257188704752804338240135223853477786261514346471147578138380300356926942242696950584180353118557970714746748293634867832256558193991251048960150789213237205376343161675566995259505494236219102538596666414291785787493146842946263424534445742459808283865087788006440187741365423244015665596823

The problem is RSA with a given value of n(product of any two large prime numbers), e(exponent), and c(cypher).

RSA Encryption

RSA algorithm is an asymmetric cryptography algorithm. Asymmetric means that it works on two different keys i.e. Public Key and Private Key. As the name describes that the Public Key is given to everyone and the Private key is kept private.

The idea of RSA is based on the fact that it is difficult to factorize a large integer. The public key consists of two numbers where one number is the multiplication of two large prime numbers. And private key is also derived from the same two prime numbers. So if somebody can factorize the large number, the private key is compromised.

The Encryption process of RSA is

$C = M^e mod N$, where e is the exponent

The Decryption process of RSA is

$M = C^d modN$, where $d = e^{-1} mod(\phi(n))$,

One possible way to break this RSA is to factorize the n but since n is very large (1024 bits), we can also try computing d but for this, we need to find $\phi(n)$ which is not easy, so we need another way of breaking this RSA.

Breaking the RSA

It is given that the exponent of RSA is 5 which is very small, So in that case, we can use Coppersmith's algorithm to break the RSA with a small exponent.

Coppersmith gives an algorithm to find the root of the polynomial mod(N) when roots are small enough.

So to apply Coppersmith's algorithm we need to form a univariate polynomial, for this first we need to check whether some padding is used or not in cypher because generally padding is used to make the message length 1023 bits, we make a thought here that password alone can't be 1023 bits so there must be some padding,

Let, p be the padding used and x be the remaining message, after applying padding we got the polynomial equation in x as,

$$f(x) = (x + p)^e - C \ mod(N)$$

here in the above equation, C, e, p, and N are known to us and we have to solve this polynomial for x under modulo N, the roots of $f(x)$ would be our final password.

Now, finding the roots of this polynomial modulo N is difficult as x is over real numbers, but if we have a polynomial over the integers which has its roots over the integers then it would be easy to find the roots.

So, the first idea of coppersmith is to find another polynomial ‘g’ that has the same roots as our problem ‘f’ but over the integers,

$f(x_0) = 0\, mod(N) \;\; with\ |x_0| < B\ changes\ to\ g(x_0) = 0\ over\ Z\ with\ ||g(xB)|| < \frac{N^{1/e}}{\sqrt{n}}$, where B is the Bound

It is also hard to generate the polynomials which have the same root as our problem, but we could generate all the polynomial that has the same root but under $mod(N^{1/e})$ and this is easier.

How to do it?

we need to use the LLL(Lattice reduction) algorithm, which is a polynomial-time lattice reduction algorithm Given a basis $B = \{b_1, b_2, ..., b_d\}$ with n-dimensional integer coordinates, for a lattice L (a discrete subgroup of R_n) with $d \leq n$, the LLL algorithm calculates an LLL-reduced (short, nearly orthogonal) lattice.

The code used to find the root of the polynomial and LLL is taken from [coppersmith.sage](https://github.com/mimoo/RSA-and-LLL-attacks)

Now we need to find what padding might be used, we looked at the password screen and the password is followed by a line

“Turing: This door has RSA encryption with exponent 5 and the password is “

So we thought this might be our padding, we take this line and converted it to the binary string “bstr”. We don’t know the length of the password but we know that the root $x_0 < N^{1/e}$, thus the message can’t be longer than 205 bits.

Since we don’t know the length of the root, we need to find the correct length of root x_0 let it is $length_x$, so we can shift binary string bstr to get the correct padding $p = bstr << length_x$

The final polynomial will now become $g(x) = ((bstr << length_x) + x)^e - C\, modN$, the polynomial ring is set over $Z|nZ$

We iterated over all the 205 bits with a window of size 8 bits(1 byte) and in each iteration root(s) of $g(x)$ are calculated, roots are calculated using SageMath's built-in function which finds all the roots which are less than $N^{1/e}$ using CopperSmith's algorithm.

the root x_0 found is
0b10000110011100001011001010100000011011101101111010011
00011011110011011001011001

"0b" represents binary, so we removed these two bits. We were left with 79 bits. So to make it a multiple of 8(to convert to ASCII code), we added '0' at the start of the password.

0100001100111000010110010101000000110111011011110100110
0011011110011011001011001

after iterating over it in chunks of size 8 and converting that chunk to decimal and finding the character corresponding to that decimal value ASCII we get our final password "**C8YP7oLo6Y**".

our code is in a file named "generate_passcode.sage" which can be run on cocalc.com to get the password.

References :

[1] [Finding small roots of univariate modular equations revisited](https://link.springer.com/content/pdf/10.1007/BFb0024458.pdf)

[2] [Finding Small Solutions to Small Degree Polynomials](https://link.springer.com/content/pdf/10.1007/3-540-44670-2_3.pdf)

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.

▾ generate_passcode.sage Download

```
e = 5
C = 494058836641125962571887047528043382401352238534777862615
N = 843644437357250348644025545338262791747038934397633434388

ZmodN = Zmod(N);

p = "Turing: This door has RSA encryption with exponent 5 and the password is "
len_M = 200

bin_p = ''.join(format(ord(i), '08b') for i in p)
beta = 1
eps = beta / 7
flag = 0
for length_M in range(0, len_M+1, 4):
    P.<M> = PolynomialRing(ZmodN)
    polynomial = ((int(bin_p, 2)<<length_M) + M)^e - C
    m = ceil(1 / (polynomial.degree() * (1/7))) # beta = 1, eps = 1/7
    X = ceil(N**((1/polynomial.degree()) - (1/7)))
    polynomial = polynomial.change_ring(ZZ)

    x = polynomial.change_ring(ZZ).parent().gen()
    degree = polynomial.degree()

    lis = [] # Polynomials
    lis+=[(x * X)**j * N**(m - i) * polynomial(x * X)**i for i in range(m) for j in range(degree)]

```

```

    y = degree * m
    lattice_B = Matrix(ZZ, y) # Lattice B

    for i in range(y):
        for j in range(i+1):
            lattice_B[i, j] = lis[i][j]

    lattice_B = lattice_B.LLL() # LLL

    polynomial = 0
    for i in range(y):
        polynomial += x**i * lattice_B[0, i] / X**i #
shortest vector

    possible_roots = polynomial.roots() # Stores the
possible roots

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

    if roots:
        print("The correct root is :", bin(roots[0]))
        flag = 1
        break
if flag==0:
    print('Solution not found')
```

Assignment 6

GRADED

GROUP

Rohit kushwah
Dinkar Tewari
Deepak Raj

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