Q1 Team Name
0 Points

Group Name

Cryptosenpai

Q2 Commands
5 Points

List all the commands in sequence used from the start screen of this level to the end of the level. (Use -> to separate the commands)

enter -> jump -> jump -> back -> jump -> pull ->back -> back -> go -> wave ->
back -> thrnxxtzy -> read -> the_magic_of_wand -> c -> read -> password -> c
-> psutdydggc

Q3 Cryptosystem
10 Points

What cryptosystem was used at this level? Please be precise.'

6 - Round Data Encryption Standard
Symmetric key block cipher, 16 round Feistal structure
64 bits Block size,
Key size 56 bits,
64 bit input leads to 64 bit output. One round of
DES consists
Expansion, Key Xor, S-box permutation, Permutation box and then
Xor again with Left half.

Q4 Analysis
80 Points

Knowing which cryptosystem has been used at this level, 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.)

Data Encryption Standard is a block cipher with Block-size = 64 bits and key size = 56 bits. We are breaking a 6-round DES. For solving this, we use DIFFERENTIAL CRYPTANALYSIS. In order to break the six rounds of the given DES, we do a chosen plaintext attack . We need a 4 round characteristic to break a 6-round DES which is obtained after 4 rounds of the 5-round characteristic discussed in lecture 7. The differential iterative characteristic starts at 405c0000 04000000 and after 4 rounds gives the differential 00540000 04000000 with probability = ¼ *5/128 * ¼ * 5/128 = 0.000381 . Thus we need approximately 20/0.000381 = 52000 chosen input pairs to get assured of finding the right key.

Generating random inputs and their ciphertext-

We generate 1 lakh random inputs of 64 bits (plaintext block). For this, we used 'input_gen.cpp' that generated 1 lakh random binary strings and stored them in inputs.txt.Then, for each of the input binary strings, we take their XOR with '0000901010005000' to generate its pair.

We send random inputs of size <= 16 alphabets manually and always get a 16-letter string as output from the game . Thus, 64-bits are required to represent 16 letters i.e 4 bits for each character so only 16 alphabets can be represented distinctly. On frequency analysis of few outputs, we observe that only letters 'f,g..u' appear as output. so these binary strings are converted to 16 letter strings assuming the following mapping : f-0000 g-0001 h-0010 ... u-1111 in the ‘take_xor.cpp’ program and store these in input_pairs.txt. Now, using the input_pairs.txt, we generate a script that has all the commands to run the game passing all the generated input_pairs to the server. This was done by ‘script_build.cpp’ and the script was stored in ‘script_game.sh’. On running this script_game.sh, we store all the terminal output in ‘game_outputs.log’ and then using ‘clean_output.py’, we extracted the output pairs generated (ciphertext block for all the 1 lakh input pairs) into ‘output_pairs.txt’. To convert these output pairs to 64-bit binary string, ‘stringtobin.cpp’ was used and the 64-bit binary was stored in outputs.txt.

6 round DES decryption-

Now, we have the input plaintext (1 lakh pairs) and output ciphertext (1 lakh pairs) in the binary form. We need output xor of S box of last round (gamma1 xor gamma2) , input xor to S box of last round (beta1 xor beta2) and expanded values of R5 (alpha1,alpha2). and R5=L6.

Using outputs.txt, we run ‘differential.cpp’ that generates: final_rev_perm.txt - stores R6L6 (where R6L6 = reverse final permutation of output ciphertexts) and then R5 (=L6) is expanded by given E BIT_SELECTION TABLE to give alpha1, alpha2 betaxor.txt - stores betaxor = alpha1 xor alpha2 . gammaxor.txt - output xor (R6 xor R’6) is inverse permuted by given inverse permutation function to get (gamma1 xor gamma2)

Using the ‘alpha1.cpp’, we choose the left blocks (L6, which is equal to R5) of only alternate outputs starting from the first output, then, expand it using E BIT_SELECTION TABLE(E-Box) and store it in ‘alpha1.txt. In ‘key_freq.cpp’ , we generate the beta1 and beta2 value pairs using the given xor and whose [S(beta1) xor S(beta2) = gamma xor value] . For each permuted beta1 , beta2 is calculated as xor of beat1 with (alpha1 xor alpha2) . The pairs that satisfy [S(beta1) xor S(beta2) = gamma xor] are selected . key 6 is calculated for all beta1 values as = alpha1 xor beta1 and frequencies of all keys is stored in key_freq.txt Max frequency keys of 6-bits corresponding to 8 S-boxes : (45 51 34 40 20 62 12 52) But then running the same code for different numbers of iterations, the corresponding s3 and s4 keys kept fluctuating , so we cannot be sure about them . Thus, Key is of the form (45 51 - - 20 62 12 52) Binary of these key bits (101101 110011 XXXXXX XXXXXX 010100 111110 001100 110100) We don’t know the bits corresponding to S3 and S4 boxes and the rest 8-bits which were not selected in k6.

‘key_permute.cpp’ permutes the binary of the 48 bits of key k6 using permutation functions PC1 and PC2 (given in resources des.txt) stores the 56-bit possible variable key in permuted_key.txt. The possible key has ‘X’ corresponding to the permuted S3 and S4 blocks and the remaining 8 bits except the 48 bits are also ‘X’.

‘key_gen.cpp’ uses the permuted_key.txt and generates all the 2^{20} possibilities for the key (replacing X with 0,1 for each X generates all such possibilities) in key_possible.txt. Then, we check all these possibilities of the key by checking whether that key generates the same output that was given by the game or not. This is done using the ‘key_brute.cpp’ which outputs the final key on the terminal.

In key_brute.cpp, we had manually entered one input-output pair that was input - {0,0,0,0,0,0,0,0} output - {162, 6, 167, 107, 210, 150, 253, 83} The input corresponds to the plaintext ‘ffffffffffffffff’ and it generates the ciphertext ‘phflpmlqsholuski’ for which stringtodec.py was used to generate {162, 6, 167, 107, 210, 150, 253, 83} which is the binary string concatenation (with f being 0000, g being 0001 and so on upto ‘u’) for pairs of letters and then each of these string is transformed to corresponding decimal.

Final key :
0110111001011110011110110000000011000101001011101101011(outputted on the terminal by running key_brute.cpp)

Our password was : “tjsoknqsutfnujfljrtrnmjrsjtorois” which is converted by ‘stringtodec.py’ into {64, 215, 133, 54, 6, 100, 58, 123}{110, 117, 216, 251, 105, 25, 40, 37}

The password was a 32 letter string (128 bits) , so it was passed in 2 parts into decrypt_pwd.cpp which decrypts the password using our key and gives the decrypted output as 112 115 117 116 100 121 100 103 103 99 48 48 48 48 48 48 Among these , the trailing 48s represent zero padding and are ignored . Rest values represent ascii values. So, the password comes out to be : "psutdydggc"

Q5 Password

5 Points

What was the password used to clear this level?

psutdydggc

Q6 Code

0 Points

Please add your code here. It is MANDATORY.

Archive 2(1).zip Download

```
Large file hidden. You can download it using the button
above.
```

Assignment 4

● Graded

Group

Rumit Pingleshwar Gore
Kuruma Abhinav
VAMSEE KRISHNA KAKUMANU
View or edit group

Total Points

92 / 100 pts

Question 1
Team Name **0** / 0 pts

Question 2
Commands **5** / 5 pts

Question 3
Cryptosystem **10** / 10 pts

Question 4
Analysis **72** / 80 pts

Question 5
Password **5** / 5 pts

Question 6
Code **0** / 0 pts