**6-round DES**

The cryptosystem used was 6 round DES. And we used Differential Cryptanalysis to figure out the password. The first hint is of "2 letters per byte", knowing that the 64 bits of DES correspond to 8 bytes, the input block size should be 16. We generated a script to access the server and get outputs for our generated input pairs. From various input output pairs and the frequency analysis of output pairs, it is easy to see that the output contains letters from f-u only. Also, inputs of size 16k+r, \; 0<r<1616*k*+*r*,0<*r*<16 gave outputs corresponding to size 16k16*k* inputs. So we concluded that the cryptosystem is working in \mod 16mod16, blocksize=16. Hence, each letter must also be mapped to 4 bits, i.e., from 0-15. Then we used the differential iterative characteristic of 6 round that is "405c0000 04000000". This gives the differential ‘00540000 04000000’ after 4 rounds and has probability of approx 0.04% so we have to check about 1 lakh plaintext - ciphertext pairs. By running the inverse of IP with a short code, the corresponding xor value came out to be "0000901010005000". We generated 1 lakh input pairs having this XOR value. Then we used the code strtobits.cpp to convert these output texts into binary strings using the f-u mapping. (f-0000, g-0001 and so on) which are stored in f5\_binarised\_outputs.txt. Now we ran 7\_diff\_cryptanalysis.cpp on f5\_binarised\_outputs.txt which produces three files , it takes f5\_binarised\_outputs.txt and apply reverse final permutation and store the output in f6\_1\_alphas\_crude.txt , Then we xorred the odd and even block pairs. We define alpha as the input value to sbox in 6th round and the output value from sbox is beta and gamma is the output value of expansion box (it expands R5 which is equal to L6). .The left hand side bit is expanded and alpha1 xor alpha2 ( these alpha values correspond to two input we took) which is same as the xor’s of input values to sbox in 6th round is stored in f6\_2\_betas.txt The right hand is xorred with xorred L5 (equal to R4 which in turn will be ‘04000000’. This is then inverse permuted to give beta1 xor beta2 and is stored in betaxor.txt. Then we ran 9\_freq\_analysis\_subbox.cpp which yielded possible values of alpha1 and alpha2 from (alpha1 xor alpha2) by iterating over alpha1. Using the following the constraints: alpha2=alpha1 xor (alpha1 xor alpha2) (from alphaxor) S(alpha1) xor S(alpha2) = beta1 xor beta2 (from betaxor) Now, we calculate all possible values of key 6 = alpha1 xor gamma1 (here gamma1 is the expansion box output which when xorred with key6 gives alpha1 value ). The original key will occur more frequently than the incorrect ones.We store the frequency data in 11\_probable\_key\_generators.txt . So from f7\_max\_frequencies.txt, we take the maximum frequency index from all boxes. Key 1 2 3 4 5 6 7 8 Block Key Block Value 45 59 - - 44 58 17 63 (101101) (111011) (BBBBBB) (BBBBBB) (101100) (111010) (010001) (111111) Max. Frequency 7932 8442 - - 8646 8266 7957 8323 Now, we get 6 blocks values out of 8 blocks but we could not get the key values from S-boxes S3 and S4. So now we have knowledge of 36 bits out of 56 bits. So, we can iterate over the unknown bits to find the actual key. But first, we have to use 10\_permute\_key.cpp which expands these 48 bits to 56 and also gives the knowledge of how the bits were permuted. This gives us the key: Key XX1XX1XXX10X1X10XXX11XX11X0X0011101X00111110X11X0111X001 Now 11\_probable\_key\_generator permutes 0,1 in all the unknown locations and generates all possible keys in f8\_probable\_keylist.txt. Then 12\_key\_checker.cpp.cpp checks for a given input-output (plaintext-ciphertext) pair to find which candidate is the correct key. We used ffffffffffffffff [0,0,0,0,0,0,0,0,] as the input and smphhshnfkpqmpki [215, 162, 45, 40, 5, 171, 122, 83] as its cipher output from the game. This gave the correct key as: \textbf{Final Key}: 01101110010111100111101110000011101100111110011101110001 Thus, the last step remains, which is decrypting the ciphertext of our password: roifnotfpulpporkqihhpkmsfgimtrtu roifnotfpulppork [201, 48, 137, 224, 175, 106, 169, 197] qihhpkmsfgimtrtu [179, 34, 165, 125, 1, 55, 236, 239] For this the 13\_final\_des\_break.cpp code was used twice and the password was acquired as follows: [114 109 119 112 107 122 104 111 107 119 48 48 48 48 48 48] This clearly represents a 16 letter password encoded in ASCII. When decoded, it gave: rmwpkzhokw000000 We concluded that the zeroes was just added by the system for padding to a 16-blocksize. Our final password is: \textbf{rmwpkzhokw}**rmwpkzhokw** Which we entered on the screen to clear the level.

The order in which we executed our code:

1\_random\_generator.cpp -> f1\_random\_strings.txt

This generates 1 lakh random binary strings of 64 bits and stores them f1\_random\_strings.txt

2\_xor\_pairs.cpp f1\_random\_strings.txt -> f2\_inputs.txt

This takes in the 1 lakh input binary strings stored in f1\_random\_strings.txt and and generates 1 lakh pairs of input plaintext such that their XOR value is 0000901010005000. These 2 lakh input plaintexts are then stored in f2\_inputs.txt .

3\_make\_script\_for\_game.cpp -> 4\_script\_for\_game.sh

This takes the 2 lakh input plaintexts and some command line arguments as team name , team password , and generates a bash script named 4\_script\_for\_game.sh which we will use to automate the connection to the game server.

4\_script\_for\_game.sh -> f3\_crude\_outputs.txt 4\_script\_for\_game.sh is the script that will help to automate the procedure of giving random inputs to 6-level DES in the game at level 4 and then collect the output. It contains all the command we will send to the ssh server , we need to grep the encrypted text corresponding to the input passed , for that we first store all the outputs of the terminal to f3\_crude\_outputs.txt

5\_get\_final\_output\_pairs.py f3\_crude\_outputs.txt -> f4\_outputs.txt

5\_get\_final\_output\_pairs.py will grep all the encrypted text we require from the ssh logs and store it in f4\_outputs.txt .

6\_strtobits.cpp f4\_outputs.txt -> f5\_binarised\_outputs.txt

6\_strtobits.cppconverts the characters of f4\_outputs.txt to 4-bit binary code and stores them in f5\_binarised\_outputs.txt .

7\_diff\_cryptanalysis.cpp f5\_binarised\_outputs.txt -> f6\_2\_betas.txt , f6\_3\_gammas.txt,f6\_1\_alphas\_crude.txt

8\_clean\_alphas.cpp f6\_1\_alphas\_crude.txt -> f6\_3\_alphas\_final.txt

The reverse permuted values of output.txt were stored in f6\_1\_alphas\_crude.txt , Now 8\_clean\_alphas.cpp takes the left half of reverse permuted values and apply the expansion left side to get f6\_3\_alphas\_final.txt .

9\_freq\_analysis\_subbox.cpp -> f7\_max\_frequencies.txt

Helps to find actual key (the one with the highest frequency)

keyscheduling+10\_permute\_key.cpp -> expands and gives key with few bits unknown

11\_probable\_key\_generator.cpp -> f8\_probable\_keylist.txt creates possible key permutations and store in f8\_probable\_keylist.txt

12\_key\_checker.cpp.cpp -> outputs correct key

13\_final\_des\_break.cpp -> yields password in ascii form in 2 halves

Answer Password: **rmwpkzhokw**