desperate-intern

sebi364

22th May 2024

Vorwort

Dieses Dokument beinhaltet einen etwas grösseren Write-up für die SHC Challenge "desperate-intern", der nicht nur die Lösung rein technisch beschreibt, sondern auch den Weg zur Lösung, versucht etwas genauer zu erklären.

Diese Challenge ist wahrscheinlich das technisch komplexeste, was ich bisher während meiner Lehre gemacht habe und ich habe über 2 Wochen gebraucht, um die finale Lösung zu finden und zu implementieren, daher kann es sein, dass einige kleine Details unterwegs vergessen gegangen sind. Ich habe aber trotzdem versucht, den ganzen Lösungsprozess hier zu dokumentieren.

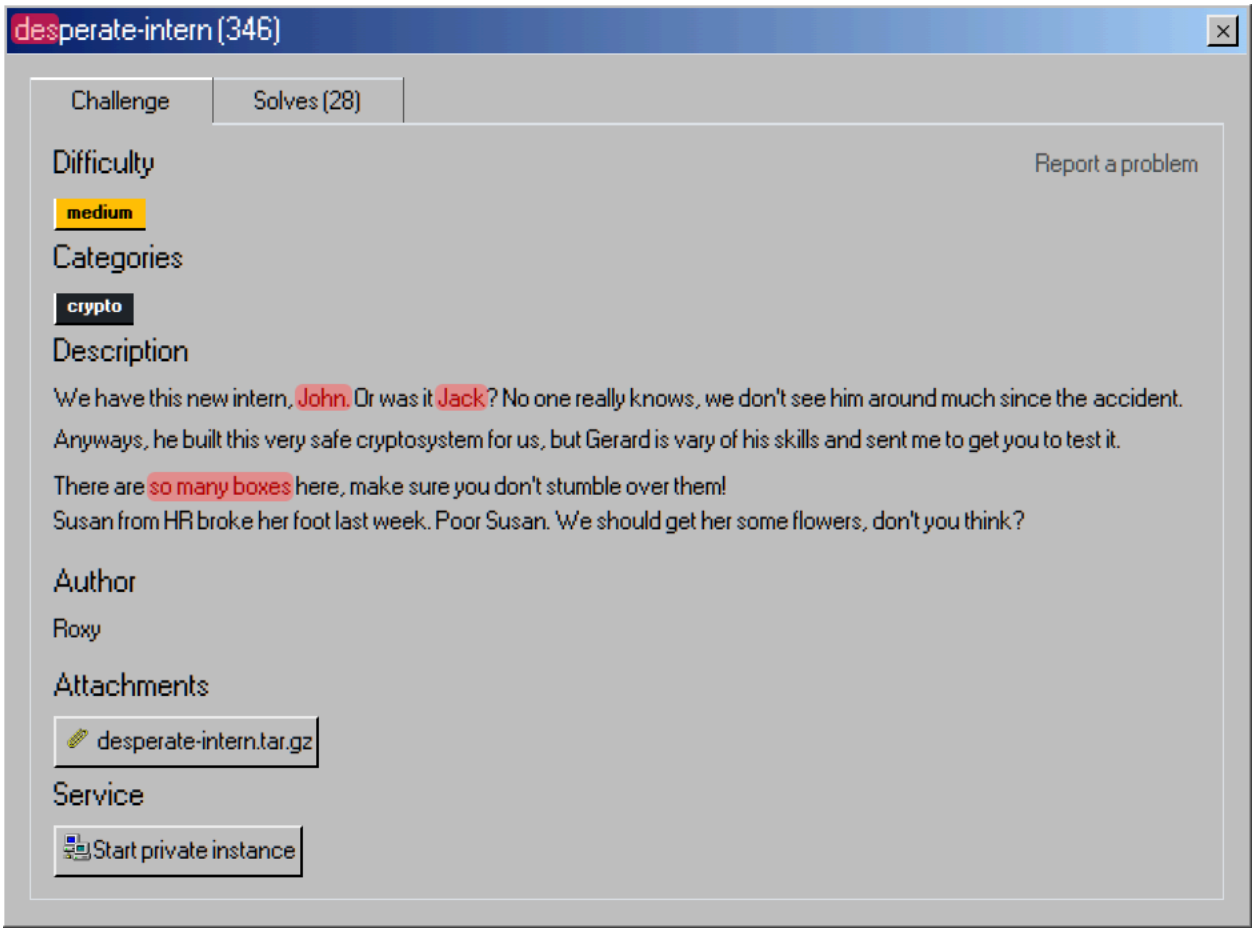
Viel spass beim lesen 🙂

Inhaltsverzeichnis

Beschreibung

Bevor wir mit einer Aufgabe anfangen, lohnt es sich immer kurz die Beschreibung anzusehen. Oft sind irgendwelche Hints & Tipps versteckt. In diesem Spezifischen fall, sah die Aufgabe Folgendermassen aus:

https://ctf.m0unt41n.ch/challenges/desperate-intern



Zusammenfassend: Der grösste Teil dieser Beschreibung macht in diesem konkreten Fall keinen Sinn, allerdings gibt es ein paar interessante Wörter, die herausstehen:

- “des”: http://en.wikipedia.org/wiki/Data_Encryption_Standard
- “John”: http://en.wikipedia.org/wiki/John_the_Ripper
- “Jack”: http://en.wikipedia.org/wiki/Jack_the_Ripper
- “so many boxes”: http://en.wikipedia.org/wiki/S-box

Sonst ist das einzige nutzvolle, was wir hier rausbekommen, dass es einen Intern gibt, der ein Verschlüsselungsverfahren entwickelt hat. **Unsere Aufgabe ist dieses Verfahren zu brechen.**

Program Funktionalität (User's POV)

Als nächstes habe ich eine Live Instanz gestartet und mich mal an die Applikation angebunden. Bevor ich anfange den Sourcecode auseinander zu nehmen, will ich mir kurz ansehen, wie die App aus der Sicht eines normalen Benutzers aussieht und funktioniert:

Wir können uns an die Laufende Instanz mit `netcat` anbinden, wo wir dann vom folgendem Menü begrüsst werden:

```
sebi@x1ng1:~$ ncat --ssl
e84838e1-a824-4612-971a-c29f5e2e.ctf.m0unt41n.ch 1337
Welcome to your homemade supersafe encryption service made by Jack the
Intern
Your credit for encrypted messages is at: 16
Would you like to get the flag [1], encrypt a message yourself [2] or
exit [3]
```

In diesem Fall scheint es so als ob wir nur 2 Optionen haben: wir können entweder eine Nachricht verschlüsseln, oder die Flag holen:

```
Would you like to get the flag [1], encrypt a message yourself [2] or
exit [3]
> 2
Enter the plaintext you want to encrypt
> windows < linux
Ciphertext =  01000000010110100...
Feistel output =  [['01111111000000100000000000010101110000110000110 1',
'00111010101101010000000001110010'],
['11100101000000101111111111001010001010010000 0111',
'00001101011010001011101010000110']]
Your credit for encrypted messages is at: 15
```

```
Would you like to get the flag [1], encrypt a message yourself [2] or
exit [3]
> 1
Flag =
000110100001100110111110010111101000010000001101110001101101001100011111
111011011011100111010000110001111001000011001011011010000110010111110011
110010000010111111100111001000001101011101110101000101001100010000010010
100000110011111001101000110001100001011011001101001101011010010111111100
011000111001100010100000001000100
Feistel output =  [['10101010100101100000100101000101111010111111010',
'00110000110100100001100010000100'],
['10111111111111000000110101100000111010101010010',
'00110110011110010000011010000110'],
['00101010110101011111011010000101100000101111010',
'00101000101101011000000001011101'],
['00100101010000011010000000011010000101101111000',
'00000101101110010001110001111111'],
['00110101101010101111110000010101001010001010000',
'10100100000111111111010000001110']]

Would you like to get the flag [1], encrypt a message yourself [2] or
exit [3]
>3
Exiting...
```

Wir können am Output des Programmes ein paar interessante Sachen erkennen:

1. Es gibt eine beschränkte Anzahl von "Credits", die wir verbrauchen können. Wir können nur **16 mal** etwas verschlüsseln.
2. Der Output der Verschlüsselungsfunktion besteht aus 2 Teilen:
    a. Dem Ciphertext ansich
    b. Einem “Feistel Output” - eine Liste mit 2 noch unbekannten Elementen
3. Die Flagge ist verschlüsselt und in der aktuellen Form komplett nutzlos.

Grober Technischer Überblick

Erster Eindruck / Python Jumpscare

Im Download der Challenge bekommen wir **dieses Python-Script**[1], das ist der Quellcode der Applikation, mit der wir vorher interagiert haben.

Bevor ich anfange das Programm in sehr genau anzuschauen, versuche ich mir einen groben Überblick zu verschaffen, je nachdem kann es sein, dass die Lösung sehr einfach ist. Hier sind ein paar Punkte die mir dabei aufgefallen sind:

Imports:

```
import secrets
from Crypto.Util.Padding import pad, unpad
...
```

In den ersten paar Zeilen werden mehrere Module importiert, aber keines der Module hat etwas gross mit Verschlüsselung zu tun.

- https://docs.python.org/3/library/secrets.html
- Pad & unpad fügen nur nuller zum schluss eines stringes hinzu damit er z.B. 8 Bytes hat: https://en.wikipedia.org/wiki/Padding_(cryptography)

Das bedeutet, dass die Verschlüsselung höchstwahrscheinlich im Code direkt implementiert ist. Generell wird nicht empfohlen[2], dass man Cryprofehrfahren selbst implementiert, wenn man nicht ganz genau weiss was man macht.

Magic Numbers

Die ersten 80 Zeilen (aka die Hälfte des ganzen Programms) sind mit Listen voller “magischer Zahlen” besetzt. Ich werde diese mal vorerst ignorieren und so tun, als ob sie nicht existieren würden.

Seed

Es ist egal, wie gut der Verschlüsselungs-Algorithmus ist, wenn der Schlüssel schwach ist und mit einer einfachen Bruteforce-Attacke gebrochen werden kann. Daher Lohnt es sich kurz zu überprüfen woher er bei dieser Implementation kommt:

---

[1] Ich habe das Script in einen Gist hochgeladen, weil er über 200 Zeilen hat und im Word Document direkt unlesbar sein würde: https://gist.github.com/sebi364/33d5edad5f5e451f1dae9b75486d2316
[2] https://crypto.stackexchange.com/questions/40805/aes-standard-c-library-implementation

```
key = int_to_64b_bitstring(secrets.randbits(64))
```

In diesem Fall besteht er scheinbar aus zufälligen 64 Bits die beim Start des Programms gewählt werden, also können wir das Programm nicht leicht von dieser Seite angreifen.

DES-ECB Mode

Nachdem ich mich ein bisschen mehr umgesehen habe, ist mir aufgefallen, dass die Hauptfunktion, die die Daten verschlüsselt, `des_ecb_encrypt` heisst. Das ist ein hinweis darauf dass das Programm eine “simple” form von DES im ECB Modus implementiert:

```
def des_ecb_encrypt(plain_text, key):
```

Crypto im Detail (pt. 1)

Verschlüsselung Zurückverfolgen

Jetzt wo wir mehr oder weniger eine grobe Idee davon haben, mit was wir arbeiten, können wir anfangen, das Programm im Detail auseinander zu nehmen und nach Sicherheitslücken zu suchen. Weil es sich hierbei um eine Crypto Challange handelt, können wir gerade bei der Verschlüsselung anfangen:

des_ecb_encrypt()

Der Anfang des gesamten Prozesses (quasi der “Entry Point”) befindet sich in der `main()` Funktion:

```
def main():
    ...
    # keygen
    key = int_to_64b_bitstring(secrets.randbits(64))
    ...
    # Example usage encrypt
    cipher_text_flag, feistel_output_flag = des_ecb_encrypt(FLAG, key)
    ...
```

Hier wird am Anfang ein neuer Key generiert, der dann später zum Verschlüsseln verwendet wird. Sonst passiert hier verschlüsselungstechnisch nicht viel, es wird nur die Funktion `des_ecb_encrypt()` aufgerufen, die können wir uns als nächstes ansehen:

```
def des_ecb_encrypt(plain_text, key):
    plain_text = pad(plain_text.encode(), 8)
    cipher_text = ''
    feistel_output = []
    for i in range(0, len(plain_text), 8):
        block = int_to_64b_bitstring(int.from_bytes(plain_text[i:i + 8],
'big'))
        curr_cipher_text, curr_feistel_output = des_encrypt(block, key)
        cipher_text += curr_cipher_text
        feistel_output.append(curr_feistel_output)
    return cipher_text, feistel_output
```

Diese Funktion sieht bereits ein bisschen komplexer aus, aber sie ist in der Praxis auch recht einfach:

1. Der Text wird zuerst encodiert, damit er nicht mehr in der “Textform” gespeichert wird, sondern als Bit-String.
2. Weil ECB Mode mit Blöcken von fester Grösse arbeitet, muss der Text eventuell noch gepaddet werden. Dies wird gemacht indem man Nuller am Ende vom Text hinzugefügt.
3. Zum Schluss wird der Text (vereinfacht) in Blöcken von 8 Byte / 8 Charakteren an die funktion `des_encrypt` zusammen mit dem Schlüssel übergeben. Diese gibt dan 2 Werte Zurück: den Ciphertext und einen `feistel_output`

des_encrypt()

Als nächstes können wir uns die `des_encrypt()` Funktion ansehen, die vermutlich für die Verschlüsselung an sich verantwortlich ist:

```
def des_encrypt(plain_text, key):
    cipher_text = ''
    round_keys = generate_round_keys_enc(key)

    plain_text = permute(plain_text, IP)
    left_half = plain_text[:32]
    right_half = plain_text[32:]

    for round_key in round_keys[:-1]:
        feistel_output = feistel_network(right_half, round_key)
        new_right_half = xor_strings(left_half, feistel_output[-1])
        left_half = right_half
        right_half = new_right_half

    feistel_output = feistel_network(right_half, round_keys[-1])
    new_right_half = xor_strings(left_half, feistel_output[-1])
    left_half = right_half
    right_half = new_right_half

    cipher_text = permute(right_half + left_half, FP)

    return cipher_text, feistel_output
```

Diese Funktion ist bereits viel komplexer, auf den ersten Blick scheint es so als ob wir sie “mental” in 3 Teile Zerlegen Können:

1. Etwas passiert Am Anfang mit dem Input der Funktion
2. Ein `for` Loop iteriert durch alle (ausser dem letzten) Keys und macht etwas mit ihnen
3. Zum Schluss passiert noch etwas mit dem Output des 2. Teil

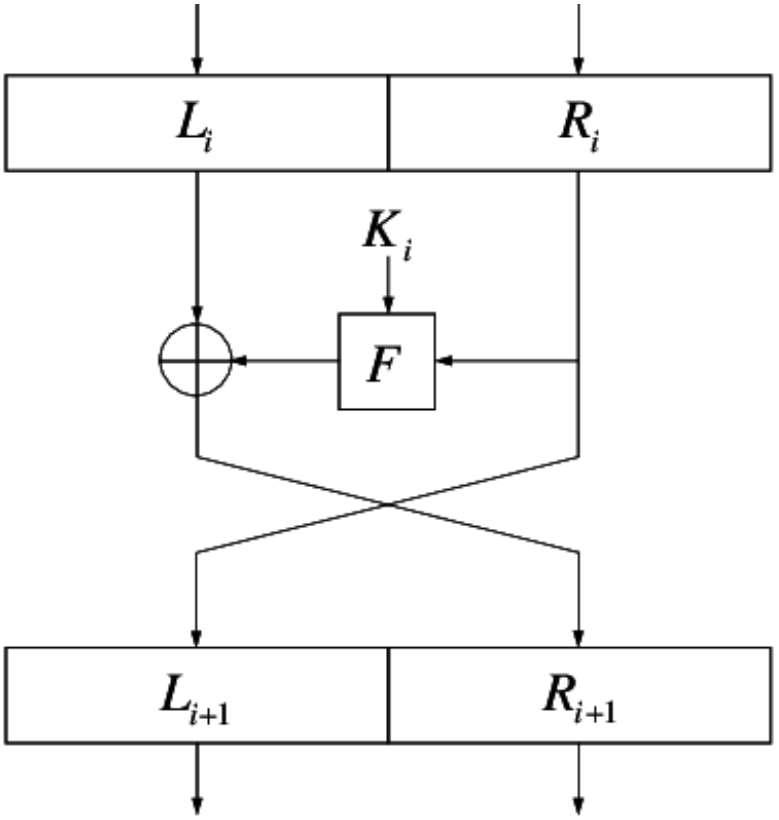
Weil es so aussieht, dass der 2. Teil das Herzstück der Verschlüsselung ist (dort passiert etwas mit den Keys), fange ich dort mit meiner Analyse an.

Wir können sehen, dass der `for` Loop durch fast alle Schlüssel in `round_keys` iteriert, diese wurden im “1. Teil” durch eine Funktion namens `generate_round_keys(key)` erstellt. Im `for` Loop drin wird dann eine weitere Funktion namens `feistel_network()` ausgeführt.

`feistel_network()` ist eine andere relativ grosse Funktion, die im Programm vorhanden ist.

Feistel Network

Nachdem ich ein bisschen recherchiert habe, habe ich herausgefunden, dass ein Feistel-Netzwerk[3] ein bekanntes Konzept in der Kryptographie ist, das dazu verwendet wird, um komplexere Cypher wie z.B. DES zu kreieren. Ein Feistel-Netzwerk Sieht Folgendermassen aus:



Input:
Zuerst wird unser Ciphertext oben in das Feistel-Netzwerk eingefügt, damit es Funktioniert muss der Text in 2 Teile unterteilt werden L_i & R_i

R_i:
Die rechte Hälfte wird in eine Funktion eingefügt (In der Grafik als F dargestellt), die den Input und den Roundkey irgendwie verwandelt, damit er sicher als XOR Key verwendet werden kann. Danach wird die Rechte-Hälfte zur neuen linken Hälfte.

L_i:
Die Linke Hälfte wird mit dem Output der Funktion F durch XOR verschlüsselt und wird danach zur neuen rechten Hälfte.

[3] https://www.conordeegan.dev/posts/feistel-network-in-python

Zusammenfügen:
Eine Runde von einem Feistel-Netzwerk kann nur jeweils eine Hälfte des Inputs verschlüsseln, daher werden immer mehrere Runden verwendet (im Falle von DES sind es 16). Jede dieser Runden hat einen eigenen Key.

Weil die Hälften nach jeder Runde "rotiert" werden, bedeutet das, dass sie sich im Endeffekt gegenseitig mit den Roundkeys verschlüsseln.

Ressourcen:
Es ist nicht super einfach ein Feistel Netzwerk in wenigen Worten zu erklären, ich musste schliesslich auch mehrere Videos & Erklärungen mir ansehen, bevor ich das Konzept richtig begriffen habe. Ich empfehle jedem, der dieses Dokument durchliest (und die Lösung auch tatsächlich verstehen möchte), die folgenden zwei Videos sich kurz anzusehen:

- Feistel Cipher - Computerphile[4]
- Feistel Cipher Structure[5]

DES

Jetzt wo wir eine grobe Idee davon haben, wie der Kernteil der Verschlüsselung funktioniert, können wir uns dem Rest zuwenden.

Bevor wir die Daten in die Feistel struktur einfügen, werden sie von folgenden Code bearbeitet:

```
def des_encrypt(plain_text, key):
    cipher_text = ''
    round_keys = generate_round_keys_enc(key)

    plain_text = permute(plain_text, IP)
    left_half = plain_text[:32]
    right_half = plain_text[32:]
    ...
```

Was passiert hier:
Zuerst definieren wir einen leeren String namens `cipher_text`, in dem wird danach der output gespeichert. Als nächstes generieren wir die 16 Roundkeys, das ist die Liste der Schlüssel, die später vom Feistel Netzwerk verwendet werden. Nachdem das erledigt ist, wird die erste "Mutation" von unserem Plaintext Input durchgeführt, dazu wird hier die `permute()` Funktion verwendet.

[4] https://youtu.be/FGhj3CGxl8I
[5] https://youtu.be/8I9xAvuGJFo

Permutation #1

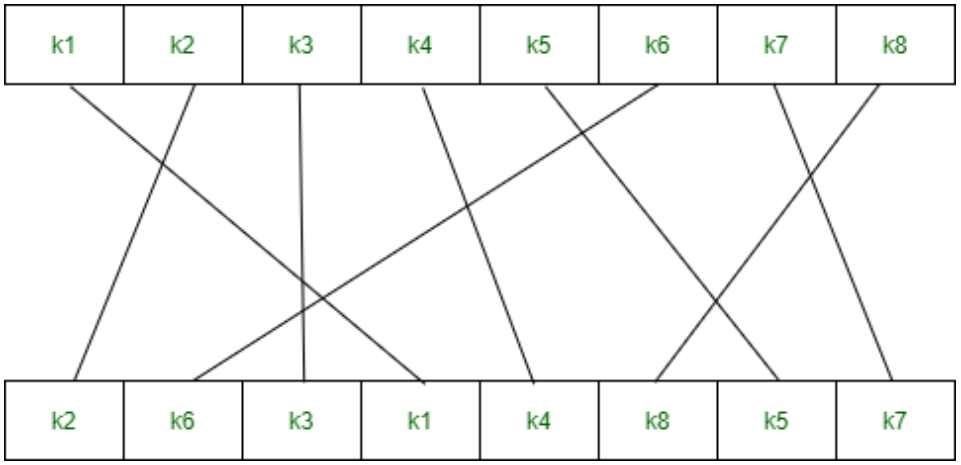
Die `permute()` Funktion sieht folgender massen aus:

```
def permute(block, table):
    res = ''
    for i in table:
        res = res + block[i - 1]
    return res
```

Die Funktion braucht 2 Inputs, den Text den wir manipulieren wollen und eine "Tabelle". In unserem Fall wird jetzt `IP` als Tabelle angegeben, das ist eine der “magischen Zahlen” die wir am Anfang ignoriert haben, und sieht so aus:

```
# Initial permutation table
IP = [58, 50, 42, 34, 26, 18, 10, 2,
      60, 52, 44, 36, 28, 20, 12, 4,
      62, 54, 46, 38, 30, 22, 14, 6,
      64, 56, 48, 40, 32, 24, 16, 8,
      57, 49, 41, 33, 25, 17, 9, 1,
      59, 51, 43, 35, 27, 19, 11, 3,
      61, 53, 45, 37, 29, 21, 13, 5,
      63, 55, 47, 39, 31, 23, 15, 7]
```

Was macht die Funktion:
Die oben gezeigte Funktion loopt durch unsere Tabelle und verschiebt die Reihenfolge des Inputs. z.B. Die erste Value in der Tabelle ist 58, das bedeutet, dass der erste Value in `res` der Charakter sein wird, der an der 58 Stelle in `block` ist. Das Endresultat nach der Permutation kann ungefähr so dargestellt werden:



Feistel Implementation

Als nächstes können wir uns nochmals kurz dem "Hauptteil" zuwenden. Im Abschnitt über Feistel Netzwerke hab ich bereits kurz erwähnt, dass wir im `for` Loop nur über alle, ausser dem Letzten Key iteriert. Der Letzte Key wird separat behandelt, das sieht im Code folgendermassen aus:

```
cipher_text = ''
round_keys = generate_round_keys_enc(key)

...

for round_key in round_keys[:-1]:
   feistel_output = feistel_network(right_half, round_key)
   new_right_half = xor_strings(left_half, feistel_output[-1])
   left_half = right_half
   right_half = new_right_half

feistel_output = feistel_network(right_half, round_keys[-1])
new_right_half = xor_strings(left_half, feistel_output[-1])
left_half = right_half
right_half = new_right_half

...

return cipher_text, feistel_output
```

Wir können hier erkennen, dass die letzte Runde des Feistel Netzwerks separat passiert. Der Grund dafür ist, dass wir von dieser Runde zusätzlich zu den beiden Hälften auch noch den `feistel_output` direkt roushollen. Wenn wir das im `for` Loop machen würden, könnten wir diese Variable nicht im `return` statement zurückgeben, weil sie nur im `for` Loop drin deklariert wurde, und daher nur dort gültig ist.

Dieser `feistel_output` wird später noch sehr wichtig sein!

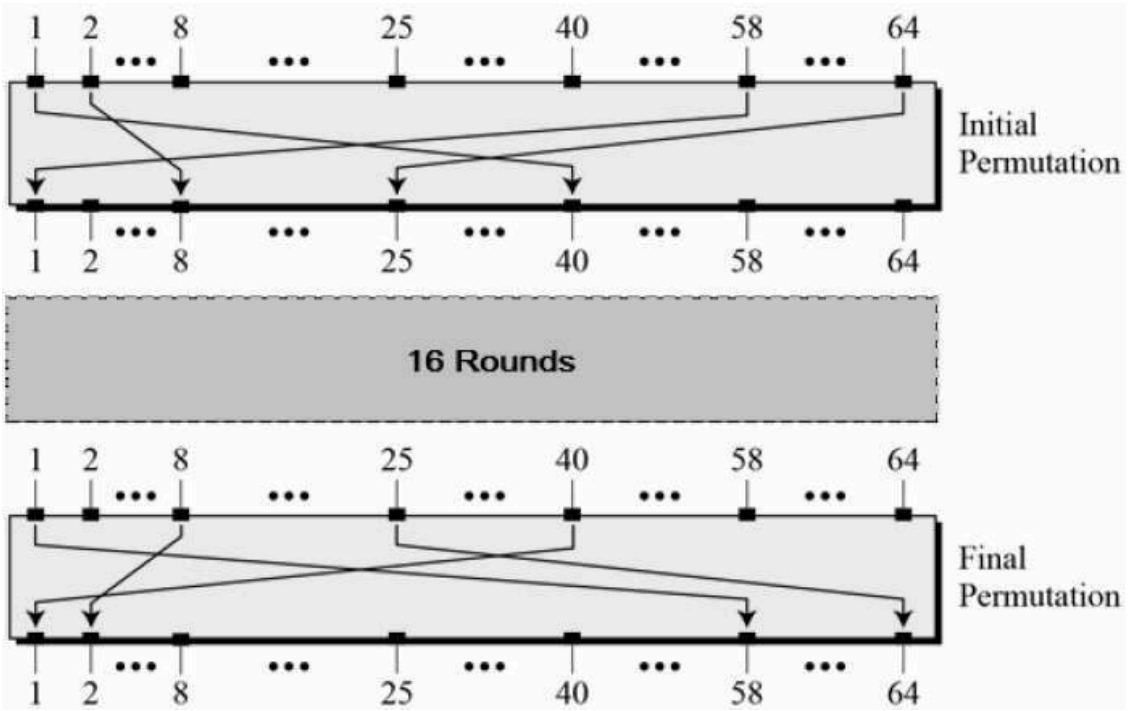
Permutation #2

Zum Schluss wird der bereits mit Feistel verschlüsselte Ciphertext nochmals permutiert, dieses mal aber mit einer anderen Tabelle namens `FP`:

```
# Final permutation table
FP = [40, 8, 48, 16, 56, 24, 64, 32,
      39, 7, 47, 15, 55, 23, 63, 31,
      38, 6, 46, 14, 54, 22, 62, 30,
      37, 5, 45, 13, 53, 21, 61, 29,
      36, 4, 44, 12, 52, 20, 60, 28,
      35, 3, 43, 11, 51, 19, 59, 27,
      34, 2, 42, 10, 50, 18, 58, 26,
      33, 1, 41, 9, 49, 17, 57, 25]


def des_encrypt(plain_text, key):
   ...

   cipher_text = permute(right_half + left_half, FP)
   return cipher_text, feistel_output
```

Zusammenfügen:

Insgesamt können die drei Teile folgendermassen in einem extrem vereinfachtn Diagramm dargestellt werden:



feistel_output

Overview

Ganz am Anfang haben wir bei der Übersicht der Applikation gesehen, dass wir zusätzlich mit dem Ciphertext auch eine Liste mit dem Namen `feistel_output` bekommen. Mit ein bisschen Backtracking können wir rasch zum Schluss kommen das es sich hierbei um den direkten Output der `feistel_network()` Funktion von der Letzten DES Runde handelt. Daher stellt sich als nächstes natürlich die Frage, aus was `feistel_output` jetzt genau besteht. Um das zu verstehen müssen wir uns die Funktion genauer ansehen:

```
def feistel_network(right_half, round_key):
    states = []
    expanded_half = permute(right_half, E)
    states.append(expanded_half)
    xored_half = xor_strings(expanded_half, round_key)
    substituted_half = substitute(xored_half)
    permuted_half = permute(substituted_half, P)
    states.append(permuted_half)
    return states
```

Hier sehen wir, dass die Funktion intern eine Liste namens `states` definiert wird. Zu dieser Liste danach wird die rechte Hälfte permutiert und hinzugefügt. Der zweite Wert besteht aus dem XOR Key. **Das ist der Key der zum verschlüsseln der letzten Runde verwendet wird!**

Letzte Runde brechen

Weil wir den Feistel-Output der letzten Runde haben, indem unter anderem auch der XOR Key der letzten Runde enthalten ist, können wir ihn dazu verwenden, um die letzte Runde zu brechen. Vielleicht können wir uns das zum Nutzen machen, um irgendwie eine Runde nach der anderen rückgängig zu machen?

Hacking Versuch #1 - (I don’t know what I'm doing)

Meine ursprüngliche Theorie war es, dass ich eventuell die Flagge entschlüsseln kann, indem ich die Letzte Runde rückgängig mache und so irgendwie versuche die Flagge *“one round at a time”* zu entschlüsseln. Ich habe gedacht, dass das die Lösung sein könnte, weil wir 16 Versuche haben, was der Anzahl von Runden entspricht. Zusätzlich habe ich Chat GPT sehr intensiv verwendet und der hat mir bestätigt, dass dies theoretisch möglich sein sollte.

Ich werde hier nicht ins Detail gehen, da diese Einstellung falsch ist und niemals funktioniert hätte. Im Grunde genommen habe ich hier einfach nur ein paar Tage verschwendet.

Hacking Versuch #2 - (Pay 2 win)

An dieser Stelle habe ich bereits mehrere Tage an dieser Aufgabe verschwendet und war dadurch ein bisschen verzweifelt. Deshalb habe ich mich dazu entschieden zu überprüfen, ob ich die Aufgabe vielleicht doch nicht einfach mit genügend Rechenleistung bruteforcen kann.

~~64 Bit~~ 56 Bit

Nachdem ich über DES ein bisschen mehr recherchiert habe, habe ich herausgefunden, dass in Wirklichkeit nur 56 Bits zum Key gehören, weil jeder 8 Bit als Checksum dient. Das macht aufgrund der exponentiellen Natur einen massiven Unterschied.

EFF's 250'000$ DES Cracker

Ich habe auch ein bisschen darüber recherchiert, ob andere Leute bereits irgendwie DES Gebrochen haben, dabei bin ich auf diesen[6] Wikipedia Artikel gestossen. In diesem Artikel wird darüber geredet, wie der EFF im Jahre 1998 einen "Supercomputer" namens Deep Crack für 250'000$ (480'000$ heute) gebaut hat, der imstande war, DES innerhalb von 56 Stunden zu Bruteforcen. Der Computer hat 1856 Custom Chips verwendet.

Nun, ich habe zwar nicht eine halbe Million rumliegen, um einen Supercomputer zu bauen, allerdings wurden seit 1998 Computer massiv weiterentwickelt und sind dadurch heutzutage viel schneller. Daher kann es ja vielleicht sein, dass ich mit ein bisschen Mühe imstande bin, den Schlüssel auf meinem Computer zu brechen.

Crack using CPU

Mein erster Versuch war, den Schlüssel direkt auf meinem Laptop zu brechen. Ich habe dafür das Programm der Challenge ein bisschen modifiziert damit er einfach jeden möglichen Schlüssel ausprobiert, um zu sehen ob man mit einem der Keys den Anfang der Flagge bekommt (`shc2024`)

Das hätte zwar theoretisch perfekt funktioniert, leider hat sich dann herausgestellt, dass ich ein paar Tausend Jahre für das Resultat warten müsste. Diese Zeit habe ich nicht, daher habe ich versucht, das Programm zu parallelisieren, damit er mehrere Versuche auf mehreren Cores gleichzeitig berechnen kann. Leider hat sich dann herausgestellt dass selbst wenn ich irgendein Programm schreiben würde, das Parallel auf allen Servern zu denen ich Zugang habe auf allen Cores gleichzeitig Läuft und wir davon ausgehen, dass alle Server die Funktion gleich schnell berechnen können, würde es immer noch sehr lange dauern.

[6] https://en.wikipedia.org/wiki/EFF_DES_cracker

Crack using GPU

Nun, es würde zwar eine Ewigkeit dauern, DES mit normalen Prozessoren zu brechen, aber was wäre, wenn wir einen GPU verwenden würden? Eine Grafikkarte ist dafür optimiert, viele ähnliche Operationen parallel zu berechnen, deshalb gibt es auch zahlreiche Bruteforce Programme, die dafür entwickelt wurden, um die Grafik-Rechenleistung zu nutzen.

Eines dieser Programme heisst Hashcat[7], und dieses kann unter anderem auch DES brechen. Weil ich selber keine super gute Grafikkarte habe, hab ich mal das Tool auf dem Computer meines Vaters gestartet, um zu überprüfen ob diese Idee realistisch ist:

```
$ nvidia-smi
Sun May 19 14:59:45 2024
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.78                 Driver Version: 550.78         CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA ****************        Off |   00000000:0C:00.0  On |                  N/A |
| 76%   73C    P2            347W / ****W |    3971MiB /  *****MiB |    100%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                              GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|    0   N/A  N/A    ******      G   ************************************       ****MiB |
|    0   N/A  N/A    ******      G   ************************************       ****MiB |
|    0   N/A  N/A    ******      G   ************************************       ****MiB |
|    0   N/A  N/A    351895      C   hashcat                                    3118MiB |
+-----------------------------------------------------------------------------------------+
```

```
Session..........: hashcat
Status...........: Running
Hash.Mode........: 14000 (DES (PT = $salt, key = $pass))
Hash.Target......: 71f1252b165a2c51:4142434445464748
Time.Started.....: Sun May 19 14:58:53 2024 (5 secs)
Time.Estimated...: Sun Jun  2 05:13:07 2024 (13 days, 14 hours)
Kernel.Feature...: Pure Kernel
Guess.Mask.......: ?1?1?1?1?1?1?1?1 [8]
Guess.Charset....: -1 DES_full.charset, -2 Undefined, -3 Undefined, -4 Undefined
Guess.Queue......: 1/1 (100.00%)
Speed.#1.........: 61354.1 MH/s (11.15ms) @ Accel:256 Loops:1024 Thr:32 Vec:1
Recovered........: 0/1 (0.00%) Digests (total), 0/1 (0.00%) Digests (new)
Progress.........: 306788171776/72057594037927936 (0.00%)
Rejected.........: 0/306788171776 (0.00%)
Restore.Point....: 0/34359738368 (0.00%)
Restore.Sub.#1...: Salt:0 Amplifier:456704-457728 Iteration:0-1024
Candidate.Engine.: Device Generator
Candidates.#1....: $HEX[73e1356573657373] -> $HEX[ffef35ffff515553]
Hardware.Mon.#1..: Temp: 69c Fan: 62% Util:100% Core:1815MHz Mem:9501MHz Bus:8
```

[7] https://hashcat.net/hashcat/

Es scheint so, als ob es tatsächlich möglich ist, DES auf einer modernen Karte innerhalb von circa 2 Wochen zu brechen, allerdings gibt es ein kleines Problem: Weil mein Vater mir die Grafikkarte seiner Workstation nicht einfach für zwei Wochen leihen kann, muss ich mir eine andere Lösung überlegen.

Nachdem ich ein bisschen studiert habe, ist mir eingefallen, dass man heutzutage Cloud GPU Server mieten kann. Diese sind zwar hauptsächlich für AI Workloads gedacht, allerdings gibt es nichts, was mich davon abhält, Hashcat auf so einem Server zu installieren.

Ich habe bei vultr.com nachgesehen, wie viel so etwas kosten würde, das ist ein VPS Provider der unter anderem auch GPU Server im Angebot hat. Dort habe ich herausgefunden, dass ein Server mit einer Nvidia A100 $3.125/Stunde ⇒ $2100/Monat kostet.

Nun, ich bin zwar ein bisschen verzweifelt, allerdings werde ich nicht hunderte von Franken für Cloud-Server ausgeben, um diese Challenge zu lösen, deshalb musste eine andere Lösung her.

Crypto im Detail (pt. 2)

Offizielle Spezifikation

Ich habe im Internet nach weiteren Dokumentationen darüber gesucht, wie DES funktioniert, dabei bin ich auf die Technische Spezifikation von DES[8] gestossen. Weil das Dokument nur 27 Seiten hat, habe ich es ganz durchgelesen. Ich habe zwar nicht alles im Detail verstanden, aber das, was ich verstanden habe, bestätigt mehr oder weniger, was ich schon bis jetzt wusste. Viel gebracht hat mir das aber nicht.

Full - Diagram

Da ich wirklich keine Idee hatte wie weiter, habe ich mich dazu entschieden ein Diagramm zu zeichnen, das den ganzen Verschlüsselungsprozess aufzeigt:

(Das diagramm ist zu gross um hier einzufügen, also hab ich es hier[9] hochgeladen)

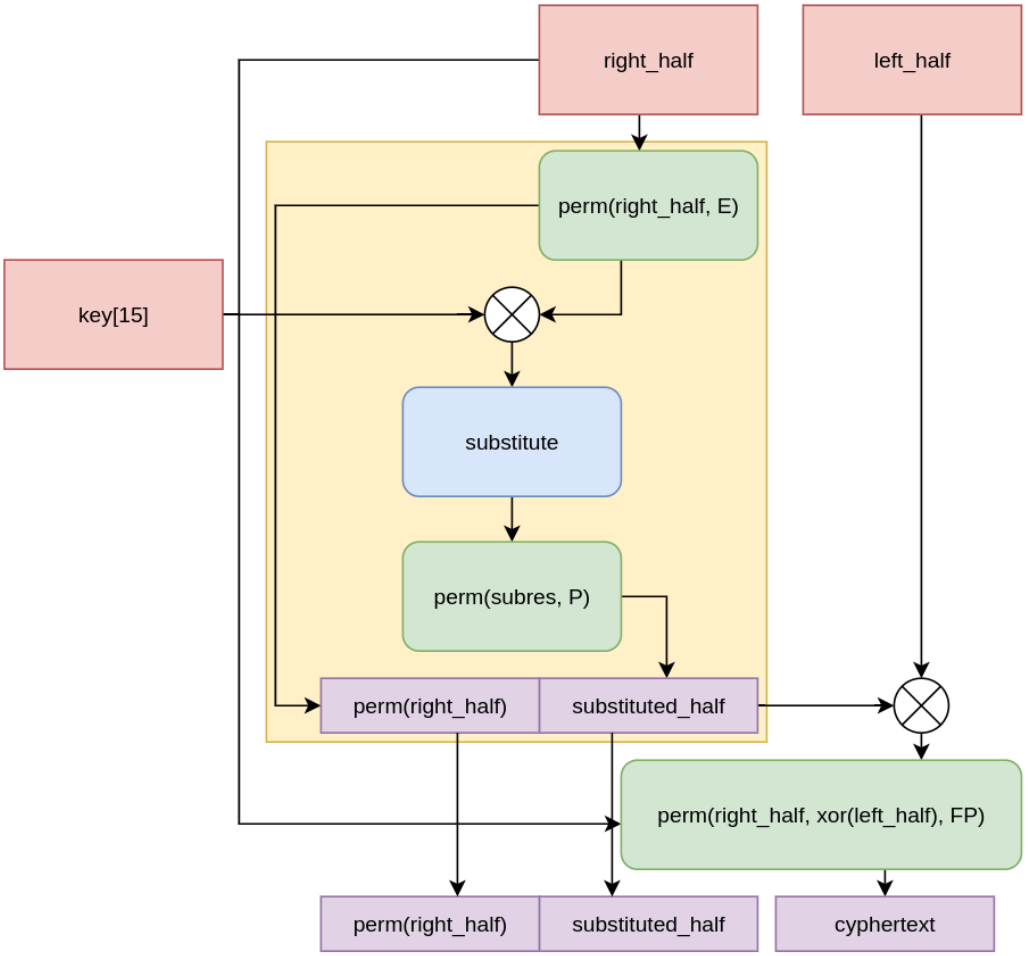
Nachdem ich dieses Diagramm in Ruhe studiert habe, habe ich verstanden, dass das, was ich probiert habe, nie funktionieren würde und der einzige mögliche potenzielle Lösungsweg ist, die Round Key-Generierung rückgängig zu machen.

[8] https://csrc.nist.gov/files/pubs/fips/46-3/final/docs/fips46-3.pdf
[9] https://drive.google.com/file/d/1_sLBCfn9UfDF7T9tf3L-aaBILrPjys68/view

Ich war (um es grosszügig zu sagen) nicht super enthusiastisch gegenüber dieser Idee, weil ich bisher davon ausgegangen bin, dass es nicht möglich ist[10], das zu machen. Allerdings war das mein einziger übriger Anhaltspunkt und ich habe schon so viel Zeit mit dieser Challenge verschwendet, dass es einen Versuch wert war.

Letzten Roundkey

Die Letzte Runde von unserer DES-Implementation sieht folgendermassen aus:



Die meisten Elemente in diesem Diagramm kennen wir bereits. Hier wird ein Feistel Netzwerk dargestellt, aber zusätzlich wird auch noch die Funktion, die den XOR Key generiert, im Detail beschrieben (gelber Kasten). Das einzige neue Konzept, was hier noch dazukommt, ist “Substitution”.

[10] Der Grund warum ich geglaubt habe das dies nicht möglich ist, ist weil ich bei meiner recherche mehrmals darauf gestossen bin das “normalerweise” bei der Roundkey generierung hashes verwendet werden. Weil ein hash by definition nicht rückgängig gemacht werden kann, würde das indirekt auch bedeuten dass die Roundkey generierung nicht rückgängig gemacht werden kann. (oder das dachte ich zumindest...)

Substitution

Die Substitution wird in diesem Code ebenfalls *“from scratch”* implementiert, die Funktion die das macht sieht folgendermassen aus:

```
...
# S-boxes (Customizable)
# Each S-box is a 4x16 matrix
# S-boxes taken from FIPS 46-3 Appendix 1
S_BOXES = [
      [0, 1, 2, 3, 4, 5, 6, 7],
      [1, 2, 3, 4, 5, 6, 7, 8],
      [2, 3, 4, 5, 6, 7, 8, 9],
      [3, 4, 5, 6, 7, 8, 9, 10],
      [4, 5, 6, 7, 8, 9, 10, 11],
      [5, 6, 7, 8, 9, 10, 11, 12],
      [6, 7, 8, 9, 10, 11, 12, 13],
      [7, 8, 9, 10, 11, 12, 13, 14]
]
...
def substitute(expanded_half):
   output = ''
   for i in range(0, len(expanded_half), 6):
      chunk = expanded_half[i:i + 6]
      row = int(chunk[:3], 2)
      col = int(chunk[3:], 2)
      val = S_BOXES[row][col]
      output += format(val, '04b')
   return output
...
```

Diese funktion ist ziemlich komplex, wir können sie aber *“step by step”* analysieren um besser zu verstehen was hier genau passiert

For loop

```
for i in range(0, len(expanded_half), 6):
   chunk = expanded_half[i:i + 6]
```

Als Erstes können wir mit dem `for` Loop anfangen. Diese iteriert über den Input der Funktion in Blöcken von 6 Zeichen auf einmal. Vereinfacht würde das so aussehen, wenn man den Input hardcodiert:

```
# Theoretischer Input
expanded_half = "110101011101"
# Effektive value beim iterieren:
for chunk in ["110101", "011101"]:
   ...
```

Weil der Input aber variieren kann, müssen diese Chunks aber “dynamisch erzeugt werden”, das macht der komische `for` Loop.

Substitution:
Als nächstes wird die eigentliche Substitution ansich durchgeführt:

```
row = int(chunk[:3], 2)
col = int(chunk[3:], 2)
val = S_BOXES[row][col]
output += format(val, '04b')
```

Dies ist in der Realität auch ziemlich einfach. Zuerst wird unser Chunk in 2 Hälften unterteilt. Die beiden Hälften werden dann jeweils in einen Integer verwandelt und als Index für die `S_BOXES` Tabelle verwendet. Wir brauchen hier 2 Indexes, weil `S_BOXES` ein 2-Dimensionaler Array ist. Die Zahl, die dann an dieser Stelle in `S_BOXES` steht, wird dann als 4 Bit dargestellt und zum Output hinzugefügt.

Wichtig!:
Es ist sehr wichtig zu verstehen, dass es sich hierbei um eine **verlusthafte Konversion** handelt. Das bedeutet, dass jeder chunk von 6 auf 4 Bits verkleinert wird. Im Endresultat ist der Output dieser Funktion nur ⅔ so gross wie der Input. Dadurch ist es auch viel schwieriger den Input der Substitutionsfunktion zu erraten, es ist aber nicht unmöglich!

Nun was können wir jetzt aber mit dieser Information anfangen? Wenn wir uns die XOR Schlüssel-Generierungs Funktion (gelber Kasten in der Grafik) anschauen, fällt auf, dass wir sowohl den Input der Funktion besitzen, als auch den Output. Wir wissen auch, dass die beiden Permutationen rückgängig gemacht werden können, daher ist es theoretisch möglich, den letzten Round Key zu bekommen, wenn wir den Input mit dem Output XORen. Damit das Aber funktioniert, müssen wir die Substitution rückgängig machen.

Unsubstitute

Um die Substitution zurückzuverfolgen, habe ich die Folgende Funktion geschrieben, sie ist “codetechnisch” nicht ein Meisterwerk, aber es funktioniert so 🙂

```
def unsubstitute(output, right_half):
    """
    Attempt to reverse the substitution process. Note that this function
    will return multiple possible keys that could have been used, because
    the same output could have been generated from multiple inputs.
    """
    expanded_half = {
        0: [],
        4: [],
        8: [],
        12: [],
        16: [],
        20: [],
        24: [],
        28: [],
    }

    # iterate over the output in chunks of 4 bits, find possible keyparts
    for i in range(0, len(output), 4):
        chunk = output[i:i + 4]
        for j in range(2**6):
            if chunk == substitute(int_to_64b_bitstring(j)[-6:]):
                expanded_half[i].append(int_to_64b_bitstring(j)[-6:])

    # create a list with all possible potential keys from the keyparts
    # this code is pure garbage, but it works so idc
    possible_keys = []
    for i in expanded_half[0]:
        for j in expanded_half[4]:
            for k in expanded_half[8]:
                for l in expanded_half[12]:
                    for m in expanded_half[16]:
                        for n in expanded_half[20]:
                            for o in expanded_half[24]:
                                for p in expanded_half[28]:
                                    possible_keys.append(xor_strings((i +
j + k + l + m + n + o + p), right_half))

    return possible_keys
```

Was passiert hier?:
Als die ursprüngliche Substitutionsfunktion den Schlüssel generiert hat, hat sie Blöcke von jeweils 6 Bits zu 4 Bits "komprimiert". Weil man mit 6 Bits nur 64 mögliche Werte darstellen kann, probieren wir einfach in diesem Code alle Möglichkeiten aus und überprüfen, welche Inputs die 4 Bits ergeben, wenn sie durch die `substitute()` Funktion gejagt werden.

Weil wir aber mehrere Chunks hatten, müssen wir diesen Vorgang für jeden 4 Bit Block wiederholen. Nachdem wir das gemacht haben, haben wir für jeden Chunk einen oder mehrere mögliche Keys. Aus diesen werden dann alle möglichen Kombinationen erzeugt, die in diesem Fall den Input korrekt konvertiert hätten.

Diese Funktion kann uns potenziell tausende von Keys zurückgeben, aber um die Anzahl der Möglichkeiten zu verkleinern, können wir den Vorgang mit mehreren Datensätzen wiederholen und die Listen der möglichen Keys aller Outputs vergleichen. Die Idee dahinter ist, dass der korrekte Schlüssel imstande sein wird, alle unsere Testdaten korrekt zu entschlüsseln, daher wird er auch in allen Outputs vorkommen.

Mit dieser Methode konnte ich die Anzahl auf circa 4 Möglichkeiten einschränken, das ist zwar nicht Perfekt, aber immer noch viel besser als wenn wir 2^{64} Schlüssel ausprobieren müssten.

Round Key Zurückverfolgen

Jetzt wo wir wissen dass der Roundkey zurückverfolgt werden kann, können wir einen gen genaueren Blick auf die Roundkey Generierungs-Funktion werfen:

```
def generate_round_keys_enc(key):
    round_keys = []
    key = permute(key, PC1)
    left_half = key[:28]
    right_half = key[28:]

    for round_num, shift_bits in enumerate(shift_table):
        left_half = rotate_left(left_half, shift_bits)
        right_half = rotate_left(right_half, shift_bits)
        round_key = permute(left_half + right_half, PC2)
        round_keys.append(round_key)

    return round_keys
```

1. Permutation

Fangen wir mal von Anfang an: Die Funktion bekommt den “echten Key” als Input, der wird dann zuerst mit `PC1` permutiert, und danach in 2 Teile unterteilt. Diese Permutation ist ein bisschen Speziell, weil wir beim Input einen 64 Bit Key haben, aber die Tabelle nur 56 Einträge hat.

```
# Permutation choice 1 table
PC1 = [57, 49, 41, 33, 25, 17, 9,
       1, 58, 50, 42, 34, 26, 18,
       10, 2, 59, 51, 43, 35, 27,
       19, 11, 3, 60, 52, 44, 36,
       63, 55, 47, 39, 31, 23, 15,
       7, 62, 54, 46, 38, 30, 22,
       14, 6, 61, 53, 45, 37, 29,
       21, 13, 5, 28, 20, 12, 4]
```

Das bedeutet, dass der Output kürzer sein wird, und bei diesem Vorgang Daten verloren gehen werden!

Die fehlenden Zahlen sind aber nicht zufällig. Im Kapitel über Bruteforce habe ich kurz erwähnt, dass jeder 8 Bit als Checksum dient und daher nicht von DES verwendet wird. Genau das können wir hier beobachten, die Zahlen für 8, 16, 24, ... (jeder 8 bit) fehlen.

Shift-Table

Als nächstes iteriert das Programm durch eine Tabelle namens `shift_table`, diese sieht folgendermassen aus:

```
shift_table = [1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1]

def rotate_left(key, bits):
    return key[bits:] + key[:bits]
```

Analysis:

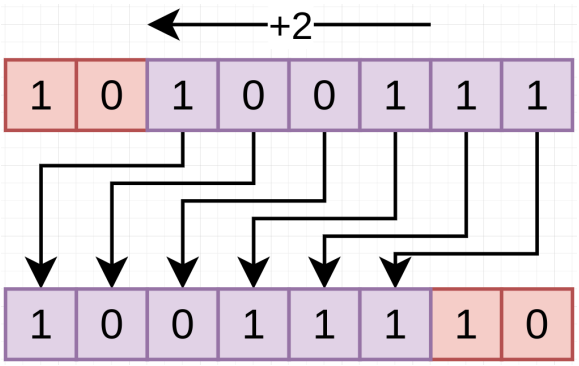
Um das Programm ein bisschen besser zu verstehen habe ich eine minimale Implementation vorbereitet, welches das Verhalten reproduziert:

```
shift_table = [1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1]

def rotate_left(key, bits):
      return key[bits:] + key[:bits]

key = "this_is_some_key"
key = "abcdefghijklmnop"
key_l = key[8:]
key_r = key[:8]

print(key_r, key_l)

for i in shift_table:
      key_l = rotate_left(key_l, i)
      key_r = rotate_left(key_r, i)
      print(key_r, key_l)
```

Der Output dieses Scripts sieht folgendermassen aus:

```
sebi@x1ng1:/tmp$ python test.py
abcdefgh ijklmnop
bcdefgha jklmnopi
cdefghab klmnopij
efghabcd mnopijkl
ghabcdef opijklmn
abcdefgh ijklmnop
cdefghab klmnopij
efghabcd mnopijkl
ghabcdef opijklmn
habcdefg pijklmno
bcdefgha jklmnopi
defghabc lmnopijk
fghabcde nopijklm
habcdefg pijklmno
bcdefgha jklmnopi
defghabc lmnopijk
efghabcd mnopijkl
sebi@x1ng1:/tmp$
```

Wir können daran erkennen dass, das Program die beiden Hälften nach rechts verschiebt. Man kann das mit einem Fliessband vergleichen, die Zahlen die links “vom Band fallen” werden rechts hinzugefügt.



2. Permutation

Jeder dieser “verschobenen Schlüssel” wird dann nochmals permutiert und dann als Roundkey verwendet.

```
def generate_round_keys_enc(key):
    ...
    for round_num, shift_bits in enumerate(shift_table):
        ...
        round_key = permute(left_half + right_half, PC2)
        round_keys.append(round_key)

    return round_keys
```

3. Realisierung

Alles, was wir bisher beim Zurückverfolgen gemacht haben, kann zu einem relativ hohen Grad rückgängig gemacht werden! Das bedeutet, dass wir imstande sein sollten, aus einem Roundkey den Ursprünglichen Schlüssel zu ziehen! Dieser kann dann dazu verwendet werden, um die Flagge zu entschlüsseln.

Hacking Versuch #3 - (This time for real)

Roundkey brechen

Als erstes müssen wir den Roundkey berechnen. Um dies zu tun habe ich die folgende Funktion geschrieben welche die in der Theorie beschriebene Attacke implementiert:

```
def get_last_round_key(cyphertext, feistel_output):
    # only take data from first DES block
    cyphertext = cyphertext[:64]
    feistel_output = feistel_output[0]
    # unpermute final permutation of cyphertext
    block = unpermute(cyphertext, FP)[:64]

    # unpermute final permutation of feistel output
    substituted_half = unpermute(feistel_output[-1], P)

    # split block into left and right halves
    right_half, left_half = cyphertext[:32], cyphertext[32:]
    print("halfes:\t\t\t", right_half, left_half)

    # unpermute right half
    substituted_half = unpermute(feistel_output[-1], P)
    print("substituted_half:\t", substituted_half)
    possible_keys = unsubstitute(substituted_half, feistel_output[0])
    return possible_keys
```

Wie bereits in der Theorie erwähnt, würde uns die `unpermute()` Funktion by Default sehr viele mögliche Matches zurückgeben. Um die Anzahl möglichkeiten einzuschränken können wir die `get_last_round_key()` Funktion mehrmals aufrufen und die Resultate vergleichen:

```
possible_keys = [
    get_last_round_key("110000...", [['1100101011...'], ...]),
    get_last_round_key("111001...", [['0110010010...'], ...]),
    get_last_round_key("111001...", [['0001010111...'], ...]),
    get_last_round_key("111001...", [['0001010111...'], ...])
]
# crossexamine possible keys to find ones that appear in every list
possible_keys_filtered = set(possible_keys[0]).intersection(
    set(possible_keys[1])).intersection(
        set(possible_keys[2])).intersection(set(possible_keys[3])
    )
print(list(dict.fromkeys(list(possible_keys_filtered))))
```

Roundkey-Generation reversen

Als nächstes müssen wir die Roundkey-Generation rückgängig machen, um an den ursprünglichen Key zu kommen. Interessanterweise, können wir dabei den Teil mit der `shift_table` überspringen. Der Grund dahinter ist, dass der letzte Key um 28 Stellen verschoben wird, was genau der Länge der Key-hälfte entspricht. Im Endeffekt dreht sich diese Hälfte einmal um sich selbst und landet wieder an der gleichen Stelle. Deshalb müssen wir uns nur um die Permutationen kümmern.

```
for key in possible_last_round_keys:
    # undo PC2 permutation
    key = unpermute(key, PC2, 56)

    # split key into left and right halves
    left_half, right_half = key[:28], key[28:]

    # undo permutation choice 1
    key = unpermute((left_half + right_half), PC1, 64)
    print(key)
```

Weil `PC1` die Ursprüngliche Tabelle verkleinert hat (bei dem Daten verloren gegangen sind), müssen wir für das bei unpermutieren von `PC1` kompensieren. Das mache ich indem ich sämtliche fehlende Stellen mit einem “`x`” ersätze.

```
def unpermute(permuted_block, table, original_len):
    key = ['x' for i in range(original_len)]
    for val_permuted, position in zip(permuted_block, table):
        key[position - 1] = val_permuted

    return ''.join(key)
```

Rückblickend ist mir jetzt beim schreiben der Dokumentation aufgefallen, das ich mir das unpermutieren von `PC1` sparen konnte. Die Restlichen Bits sind eigentlich nur Checksummen und wir können eigentlich mit den Roundkey schon nachdem wir ihn mit `PC2` unpermutiert haben dazu verwenden um die Restlichen keys zu generieren 😓. Das hätte auch den nächsten Schritt erleichtert.

Bruteforce

Wir konnten zwar den grössten Teil des Keys reversen, allerdings haben wir immer noch ein paar Stellen, die nicht rückgängig gemacht werden konnten. Zusätzlich haben wir immer noch 4 mögliche Schlüssel, weil wir uns nicht ganz genau sicher waren, was der Roundkey ganz genau war. Glücklicherweise haben wir schon einen genug grossen Teil vom Schlüssel, dass wir den Rest innerhalb eines sinnvollen Zeitraums mit Bruteforce erraten können:

```
import itertools
from des_server_handout import des_encrypt, int_to_64b_bitstring

KEY_TEMPLATES = [
    "1111110x11010xxx10xx000x1001000x1100001x1010010x11x11x0x0x1x111x",
    "1111110x11010xxx10xx000x1001000x1100001x1010010x11x11x1x0x1x011x",
    "1111010x11010xxx10xx000x1001000x1100001x1010110x11x11x1x0x1x011x",
    "1111010x11010xxx10xx000x1001000x1100001x1010110x11x11x0x0x1x111x",
]

cyphertext = "10111111111001001001100111100111110100010110000101..."[:64]

for template in KEY_TEMPLATES:
    print(template)
    unkown_bits = template.count("x")
    for bits in itertools.product("01", repeat=unkown_bits):
        missing_bits = template.replace("x", "{}").format(*bits)
        key = template.replace("x", "{}").format(*bits)
        plain_text = int_to_64b_bitstring(
                        int.from_bytes("shc2024{"[:8].encode(),
                                       'big'))
        encrypted = des_encrypt(plain_text, key)[0]

        if encrypted == cyphertext:
            print(key)
            exit(0)
```

Flagge entschlüsseln

Zum schluss müssen wir nur noch eine Funktion implementieren die den Schlüssel auch dazu benutzen kann um etwas zu entschlüsseln:

```
def des_decrypt(cypher_text, key):
    round_keys = list(reversed(generate_round_keys_enc(key)))

    plain_text = permute(cypher_text, IP)

    right_half, left_half = plain_text[32:], plain_text[:32]

    for round_key in round_keys:
        feistel_output = feistel_network(right_half, round_key)
        right_half, left_half =
              xor_strings(left_half, feistel_output[-1]), right_half
    out = permute(right_half + left_half, FP)
    return out
```

Und voila! Schon haben wir die Flagge:

```
shc2024{Apes_Dont_Read_Philosophy}
```

Rückblick

Ich habe bei diese Challenge extrem viel über Kryptographie gelernt und verstehe jetzt dadurch auch, wie DES im Detail auf einer mathematischen Ebene funktioniert.

Rückblickend sieht die Lösung eigentlich ganz logisch aus und ich verstehe selber nicht, warum ich mir die Roundkey-Generierung nicht vorher angesehen habe. Das ist wahrscheinlich ein bisschen ein Side-Effect davon das ich nicht sehr viel Erfahrung mit CTF's habe, deshalb bin ich einfach für eine weile herumgeirrt bis ich alle ausser eine Methode ausgeschlossen habe. Dieses Vorgehen hat zwar irgendwie dieses Mal funktioniert, aber ich sollte in der Zukunft strategischer vorgehen.

Zusätzlich sind einige Sachen die ich probiert habe richtig dumm gewesen. z.B. es hätte eigentlich klar sein sollen, dass die Autoren wahrscheinlich die Aufgabe so geschrieben haben, dass man nicht eine massive Anzahl von Server Ressourcen braucht, um sie zu lösen. Ich hätte viel früher realisieren sollen, dass ich irgendetwas falsch gemacht habe, wenn ich an Berechnen bin *"wie viele Wochen es dauern würde, wenn ich die Aufgabe (rein theoretisch) auf mehreren Servern mit 128 Core zu Brute-Forcen versuchen würde"*.

[Off-Topic] Kommentar zu Coden mit Copilot & ChatGPT

Das ist zwar ein bisschen Off-Topic, aber ich möchte diese interessante Erfahrung / Opinion trotzdem hier teilen weil ich denke das es zu einem Sehr aktuellen Thema ist:

Ich habe seit circa einem halben Jahr Github Copilot täglich zum Programmieren & Schreiben von simplen Scripts verwendet, was immer super funktioniert hat. Mein Rational dahinter war: Ich weiss ja wie das funktioniert, daher ist es ok, dass ich ChatGPT verwende, um meine Arbeit "zu beschleunigen". Solange ich nicht einfach alles mit ChatGPT generiere, sondern auch selbst neue Sachen lerne, sollte das kein Problem sein, im Gegensatz zu meinen Schulkollegen, die seit dem 1. Lehrjahr alles mit ChatGPT (versuchen zu) lösen.

Leider muss ich rückblickend sagen, dass ich denke, dass ich mich dabei geirrt habe. Ich habe angefangen Copilot zu intensiv zu verwenden, was in 99% der Fälle kein Problem war, allerdings hab ich mich dadurch bei komplexen Aufgaben (wie dieser hier) oft mir selbst in den Fuss geschossen. Statt selber zu überlegen, öffne ich einfach intuitiv den AI Assistant und versuche, ihm die Frage zu stellen. Zusätzlich habe ich gemerkt, dass ich manchmal mit einigen Grundlagen Probleme habe, weil ich mich durch Copilot davon abgewöhnt habe.

Deshalb habe ich mich dazu entschieden, Copilot nicht mehr zu verwenden. Ich denke, dass das langfristig wahrscheinlich besser für mich ist, auch wenn ich wieder selber etwas mehr tippen werden muss.