Google CTF 2018 Quals - chq-matteo @ mHACKeroni

DM COLLISION

DIFFIE-HELLMAN FIXED POINTS

Overview

There are two python3 scripts in the challenge attachment:

1. challenge.py
2. not_des.py

File challenge.py

There are two interesting function in challenge.py

Challenge(flag, reader, writer)

```
def Challenge(flag, reader, writer):
  try:
    b1 = Compress(reader)
    b2 = Compress(reader)
    b3 = Compress(reader)

    if b1.key + b1.input == b2.key + b2.input:
      writer.write(b'Input blocks should be different.')
      writer.flush()
      return 1

    if b1.output != b2.output:
      writer.write(b'No collision detected.')
      writer.flush()
      return 1

    if b3.output != [0] * BLOCK_SIZE:
      writer.write(b'0 pre-image not found.')
      writer.flush()
      return 1

    writer.write(flag)
    writer.flush()
    return 0

  ...
```

It takes three outputs of the Compress function (b1, b2, b3) and checks that:

1. b1 and b2 have different (input, key) pairs, but the same output
2. that b3 output is an 8 byte string of nulls ('\0')

Passing the checks will earn you the flag.

Compress(reader)

```
def Compress(reader):
  """Davies-Meyer single-block-length compression function."""
  key = reader.read(KEY_SIZE)
  inp = reader.read(BLOCK_SIZE)
  output = Xor(DESEncrypt(inp, key), inp)
  return Block(key, inp, output)
```

Since KEY_SIZE and BLOCK_SIZE are defined in not_des.py and are equal to 8, key and inp are 8 byte strings.

The function docstring gives away that we have to look for the Davies-Meyer compression function. Searching for "Davies-Meyer compression function" I found the Wikipedia page for One way compression function[1] (1wcf).

The gist is that desirable properties of a 1wcf are:

1. Collision-resistance: it should be hard to find two different key and input pairs that result in the same output
2. Preimage-resistance: it should be hard to find a key and input pair that results in a chosen output

So the challenge is to prove that Compress(reader) doesn't have either property.

[1] One-way compression function, https://en.wikipedia.org/wiki/One-way_compression_function

File not_des.py

It is a 266 line long script with several interesting functions and constant values.

```
def DESEncrypt(plaintext, key):
def DESDecrypt(ciphertext, key):

def KeyScheduler(key):
def CipherFunction(key, inp):

def Expand(v):
def LeftShift(v, t=1):
```

It definitely looks like DES[2] (a symmetric encryption algorithm).

Before committing to reverse such a long script, let's overlook the file name (not_des) and conjecture that there aren't major differences between this implementation and DES.

[2] DES, https://en.wikipedia.org/wiki/Data_Encryption_Standard

Solution

Part 1

During the overview of the challenge we found that to get the flag we have to find a collision for the Compress(reader) function

```
if b1.key + b1.input == b2.key + b2.input:
  writer.write(b'Input blocks should be different.')
  writer.flush()
  return 1

if b1.output != b2.output:
  writer.write(b'No collision detected.')
  writer.flush()
  return 1
```

The output of each block is computed as follows

```
output = Xor(DESEncrypt(inp, key), inp)
```

The most straightforward way would to find them would be trying to generate random keys or messages until we find a collision. Since output is an 8 byte string we would expect to find such collision with high probability (more than 75%) with just 7.2 billions-odd trials[3].

If you have the computing power brute force would be a no-brain solution.

Instead let's think of some well known DES pitfall, hopefully we will find a shortcut:

1. weak-keys, keys for which encrypting a message twice yields the initial message

[3] For more information on how you get these numbers https://en.wikipedia.org/wiki/Birthday_attack

2. semi weak-keys, pair of keys for which encrypting with a key is the same as decrypting with the second key
3. the keys are 64 bit long but **only 56 bit are used in the actual algorithm**

Actually **each of these properties** can be used to find a collision.

Finding a collision

```
output = Xor(DESEncrypt(inp, key), inp)
```

Notice that the Xor operation does a bitwise xor of inp and inp encrypted with key.

Since the Xor operation is commutative, **the order of its arguments does not matter**.

Solution 3: redundant bits

There are 8 unused bit of key material which are used as integrity (parity) bits for the key.

In this implementation, however, **these bits are not checked**. Every least significant bit of every key-byte is discarded in DESEncrypt.

Thus sending

- b'\0\0\0\0\0\0\0\1' b'deadbeef'
- b'\0\0\0\0\0\0\0\0' b'deadbeef'

clears the first part of the challenge.

Other solutions

Solution 1 and Solution 2 are left as exercise for the reader.

Part 2

For the second part of the challenge we need to find the preimage of a 8 byte long string of 0 bytes. Generally that would require a complete search of the input space which is 8 bytes for the input and 8 bytes for the key, not quite feasible for today's hardware.

Reframing the problem

```
output = Xor(DESEncrypt(inp, key), inp)
```

For the output of the compression function to be 0, the return value of DESEncrypt(inp, key) must be equal to inp. This is a property of the Xor function.

An argument inp for which DESEncrypt(inp, key) == inp is called a *fixed point* of DESEncrypt. Fixed points are a topic of major interest in cryptography, optimization and other problems.

That is there should be plenty of published research involving fixed points in DES.

The first result for 'des fixed points' on Google search is What is the fixed point attribute of DES (when used with weak-keys) on StackExchange which puts us on the right track. The author of the post says that during a lecture he found out that there are 2^{32} fixed points for each weak key in DES. This information reduces the search space significantly, to 8 bytes of inp since we can fix the 8 byte key.

Since there are 2^{32} fixed points we expect to find one in 2^{34} trials with very good probability!

That is well within one day of cpu time.

However we are too impatient to wait one day for the flag, let's check the two paper linked in the StackExchange comment section.

The first paper

The Real Reason for Rivest's Phenomenon explains some of the properties of these 2^{32} fixed points.

In DES there are 16 round keys which are derived from the main key. Under normal circumstances those are different, but with a weak key, each of these 16 round keys has the same value. Due to the structure of the algorithm (called feistel network) **we can know after only 8 rounds if an input is a fixed point or not, resulting in a near 2x speedup**.

The second paper

Cycle Structure of the DES with Weak and Semi-Weak Keys examines some short cycles in DES.

We actually find **some fixed point examples at page 21**.

Sadly those **do not work**!

All that glitters is not DES

We can now reasonably convince ourselves that this is not textbook DES.

If you haven't already, you should pause and understand how does DES work first.

Let's look into it in depth:

```
SBOXES = [S6, S4, S1, S5, S3, S2, S8, S7]
```

The order of the SBOXES is different

```
KS_SHIFTS = [1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1]
```

Key shift values are different

Everything else seems to be in order.

Solution

We are left with no choices other than brute forcing the input. In order to get an answer as fast as possible we employ the partial result trick and use a C implementation of DES rather than the python3 one we are given.

After 3 hours we get a collision and the flag

Appendix

Final fixed point finder

https://github.com/chq-matteo/DES

Patches applied to reference DES

https://github.com/chq-matteo/DES/commit/41d2ae5bb5e1ec2d96a0a7219cb677d6a990fa39

Final script to get the flag

```
import pwn
from binascii import unhexlify,hexlify

with pwn.remote('dm-col.ctfcompetition.com', 1337) as r:
    r.send(unhexlify('011F011F010E010E'))
    r.send('\x00'*8)
    r.send(unhexlify('1F011F010E010E01'))
    r.send('\xaf_6\xe3\xbf\xe0p\xbd')
    r.send(unhexlify('0'*16))
    r.send(unhexlify('f62470496058f670'))
    r.interactive()
```

Flag

CTF{7h3r35 4 f1r3 574r71n6 1n my h34r7 r34ch1n6 4 f3v3r p17ch 4nd 175 br1n61n6 m3 0u7 7h3 d4rk}

Expected number of trials

The probability to find one fixed point with one random trial is $1/2^{32}$, conversely the probability to not find one is pn = $(2^{32} - 1) / 2^{32}$. The probability of not finding any fixed point in n = 2^{34} trials is pn^n = 0.018315638880205296, that is less than 2 percent.