Solution for the HTB challenge Always Has Been.

This solution applies diferential analysis over the hash function, maping output differences to input differences.

Files: Drive Link

HINT: Q29sbGlzaW9ucw==

In [0]:

```python
import matplotlib.pyplot as plt
from matplotlib import colors
import numpy as np
from numpy.core.multiarray import unpackbits
import math

KEY_SBOX = [170, 89, 81, 162, 65, 178, 186, 73, 97, 146, 154, 105, 138, 121, 113, 130,
33, 210, 218, 41, 202, 57, 49, 194, 234, 25, 17, 226, 1, 242, 250, 9, 161, 82, 90,
169, 74, 185, 177, 66, 106, 153, 145, 98, 129, 114, 122, 137, 42, 217, 209, 34, 193,
50, 58, 201, 225, 18, 26, 233, 10, 249, 241, 2, 188, 79, 71, 180, 87, 164, 172, 95,
119, 132, 140, 127, 156, 111, 103, 148, 55, 196, 204, 63, 220, 47, 39, 212, 252, 15,
7, 244, 23, 228, 236, 31, 183, 68, 76, 191, 92, 175, 167, 84, 124, 143, 135, 116, 151,
100, 108, 159, 60, 207, 199, 52, 215, 36, 44, 223, 247, 4, 12, 255, 28, 239, 231, 20,
134, 117, 125, 142, 109, 158, 150, 101, 77, 190, 182, 69, 166, 85, 93, 174, 13, 254,
246, 5, 230, 21, 29, 238, 198, 53, 61, 206, 45, 222, 214, 37, 141, 126, 118, 133, 102,
149, 157, 110, 70, 181, 189, 78, 173, 94, 86, 165, 6, 245, 253, 14, 237, 30, 22, 229,
205, 62, 54, 197, 38, 213, 221, 46, 144, 99, 107, 152, 123, 136, 128, 115, 91, 168,
160, 83, 176, 67, 75, 184, 27, 232, 224, 19, 240, 3, 11, 248, 208, 35, 43, 216, 59,
200, 192, 51, 155, 104, 96, 147, 112, 131, 139, 120, 80, 163, 171, 88, 187, 72, 64,
179, 16, 227, 235, 24, 251, 8, 0, 243, 219, 40, 32, 211, 48, 195, 203, 56]
PBOX = [59, 82, 101, 135, 189, 153, 105, 14, 179, 71, 167, 33, 160, 198, 218, 104, 66,
37, 216, 199, 132, 214, 217, 42, 231, 221, 236, 233, 203, 24, 220, 120, 158, 240, 84,
81, 152, 201, 57, 253, 249, 169, 79, 234, 136, 12, 40, 209, 29, 224, 17, 77, 60, 102,
195, 8, 212, 95, 147, 190, 138, 213, 98, 10, 4, 243, 1, 128, 145, 58, 241, 119, 88,
211, 110, 157, 3, 188, 19, 208, 44, 244, 122, 92, 109, 69, 134, 22, 90, 61, 202, 193,
141, 183, 133, 75, 144, 116, 191, 39, 207, 140, 192, 247, 83, 43, 121, 99, 254, 226,
177, 26, 9, 173, 78, 176, 223, 210, 156, 16, 227, 125, 93, 54, 76, 150, 5, 36, 185,
65, 72, 246, 131, 41, 106, 248, 151, 182, 204, 225, 229, 70, 7, 250, 115, 85, 163,
124, 184, 130, 239, 196, 15, 100, 252, 25, 171, 143, 0, 67, 222, 96, 165, 180, 46,
232, 117, 48, 38, 161, 50, 35, 73, 18, 154, 114, 175, 146, 148, 89, 80, 112, 228, 49,
172, 63, 123, 86, 149, 103, 230, 64, 28, 27, 166, 111, 170, 55, 47, 20, 51, 215, 32,
13, 118, 11, 53, 205, 238, 91, 6, 94, 200, 181, 162, 178, 194, 126, 164, 2, 255, 137,
242, 23, 74, 197, 142, 108, 52, 187, 129, 186, 155, 97, 107, 34, 245, 68, 56, 127, 21,
219, 159, 62, 113, 237, 206, 45, 251, 168, 87, 31, 30, 235, 174, 139]
BLOCK_SIZE = 32

xor = lambda a,b: bytes([b1 ^ b2 for b1, b2 in zip(a,b)])
to_np_array = lambda a: np.array(list(a)).astype(np.ubyte)
to_bit_array = lambda a: unpackbits(to_np_array(a))
normalized = lambda a: (a - np.min(a)) / (np.max(a) - np.min(a))
to_bytes = lambda a : bytes(np.packbits(np.where(normalized(a) > 0.5, 1, 0)))
```

In [0]:

```python
def plot_data(data):
    assert len(data) == 32
    print("\n> Hex:", data.hex(), "\n")
    data = to_bit_array(data)
    data = np.reshape(data,(16,16))
    # create discrete colormap
    cmap = colors.ListedColormap(['black', 'white'])

    bounds = [0,0.5,1]
    norm = colors.BoundaryNorm(bounds, cmap.N)

    fig, ax = plt.subplots()
    ax.imshow(data, cmap=cmap, norm=norm)

    # draw gridlines
    plt.show()
    print("[0 -> Black] :: [1 -> White]\n")

def plot_data_scaled(data):
    assert len(data) == 256
    data = np.reshape(data,(16,16))

    plt.rcParams["figure.figsize"] = [7.50, 3.50]
    plt.rcParams["figure.autolayout"] = True

    plt.gray()
    plt.imshow(data)
    plt.show()
```

In [0]:

```
TEST_DATA = {'plain': bytes('01234567890123456789012345678901', 'utf-8'), 'hash':
'4d31d6e3e89b2510e7a6f51c9aed49ea5ca82d8d6d1e4873b695a99eab865c28'}

#substitution
def S(data):
    return bytes([KEY_SBOX[b] for b in data])

#permutation
def P(data):
    out = [0]*32
    for num in range(256):
        outnum = PBOX[num]
        inbyte = num // 8
        inbit = 7 - (num % 8)
        outbyte = outnum // 8
        outbit = 7 - (outnum % 8)

        if data[inbyte] & (1 << inbit):
            out[outbyte] |= (1 << outbit)
    return bytes(out)

def hash(data, iters = 100):
    assert len(data) == 32
    k0 = bytes([data[0]] * 32)
    key = data
    #return xor(xor(P(S(data)), P(k0)), data)
    for _ in range(iters):
      data = xor(xor(P(S(data)), P(k0)),key)

    return data

assert hash(TEST_DATA['plain']).hex() == TEST_DATA['hash']
```

In [0]:

```
from tqdm import tqdm
```

```
def get_dif(prev, bit):
  byte = bit // 8
  bit = bit - (byte*8)
  tmp = bytearray([0] * 32)
  tmp[byte] = (1 << 7-bit)
  diff = hash(xor(prev,tmp))
  prev = hash(prev)
  diff = xor(prev,diff)
  return to_bit_array(diff)

def generate_diff_matrix():
  var = []
  total = np.ndarray(256, dtype=np.uint)
  for bit in range(256):
      total = get_dif(np.random.bytes(32), bit)
      for _ in tqdm(range(100)):
        total = total + get_dif(np.random.bytes(32), bit)
      plot_data_scaled(total)
      var.append(total)
  with open('backup.npy', 'wb') as f:
    np.save(f, np.array(var))

generate_diff_matrix()
```

In [0]:

```
def get_zipped_array(filename):
  arr = np.load(filename)
  bytes_arr = []
  for each in arr:
    bytes_arr.append(to_bytes(each))

  n = len(bytes_arr)
  arr = [0]*n
  data_arr= [int(math.pow(2, n - i - 1)) for i in range(n)]
  arr = [int.from_bytes(bytes_arr[i], 'big') for i in range(n)]

  return list(zip(arr, data_arr))

#for diff in diff_bytes:
#  print(diff.hex())

diff_arr = get_zipped_array('backup.npy')
#for diff in diff_arr:
  #print('{0:256b}'.format(diff[1]), diff[0].to_bytes(32, 'big').hex())
  #print(diff[1].to_bytes(32, 'big').hex(), diff[0].to_bytes(32, 'big').hex())


t_key = bytes('01234567890123456789012345678901', 'utf-8')
out = hash(t_key)

assert hash(bytes([64]+[0]*31)) == xor(hash(bytes([0]*32)), diff_arr[1]
[0].to_bytes(32, 'big'))

for diff in diff_arr:
  new_key = xor(t_key, diff[1].to_bytes(32, 'big'))
  new_out =  xor(out, diff[0].to_bytes(32, 'big'))
  assert hash(new_key) == new_out
```

In [0]:

```
def setBitNumber(n):
    n |= n>>1
    n |= n>>2
    n |= n>>4
    n |= n>>8
    n |= n>>16
    n |= n>>32
    n |= n>>64
    n |= n>>128

    n = n + 1
    return (n >> 1)

def get_subsets(filename):
    complete = get_zipped_array(filename)
    n = len(complete)
    for i in range(n):
      complete = sorted(complete, key=lambda pair: pair[0], reverse=True)
      msb = setBitNumber(complete[i][0])
      for j in range(i+1, n):
        if complete[j][0] < msb:
          break
        complete[j] = (complete[j][0] ^ complete[i][0], complete[j][1] ^ complete[i]
[1])

    return complete

subs_matrix = get_zipped_array('backup.npy')
assert hash(bytes([64]+[0]*31)) == xor(hash(bytes([0]*32)), subs_matrix[1]
[0].to_bytes(32, 'big'))
complete = get_subsets('backup.npy')

t_key = bytes('01234567890123456789012345678901', 'utf-8')
out = hash(t_key)

#for diff in complete:
#  print('{0:256b}'.format(diff[0]), diff[1].to_bytes(32, 'big').hex())

cero_hash = hash(bytes([0]*32))
for diff in complete:
  new_key = diff[1].to_bytes(32, 'big')
  new_out =  xor(cero_hash, diff[0].to_bytes(32, 'big'))
  assert hash(new_key) == new_out
```

In [0]:

```
def get_collision(key):
  return xor(key, complete[-1][1].to_bytes(32, 'big'))

key = np.random.bytes(32)
assert hash(key) == hash(get_collision(key))
```

In [0]:

```
cero_hash = hash(bytes([0]*32))
def hash_matrix_based(key, filename):
  subs_matrix = get_zipped_array(filename)
  key = int.from_bytes(key, 'big')
  h = 0
  for i in range(256):
    if(key & (1 << 255-i)):
      h ^= subs_matrix[i][0]
  return xor(cero_hash, h.to_bytes(32,'big'))

key = np.random.bytes(32)
assert hash(key) == hash_matrix_based(key, 'backup.npy')
```

In [0]:
```
def split_target(target, filename):
  cero_hash = hash(bytes([0]*32))
  target = int.from_bytes(xor(cero_hash, target), 'big')
  subs_matrix = get_subsets(filename)

  step = 0
  key = 0
  for i in range(0,256):
    if setBitNumber(subs_matrix[i-step][0]) < (1 << 255 - i):
      step += 1
      continue
    if target & (1 << 255 - i):
      target ^= subs_matrix[i-step][0]
      key ^= subs_matrix[i-step][1]
  return key.to_bytes(32,'big')
```

In [0]:
```
target =
bytes.fromhex('61b5649e894a15a053276c0dc828ee64ec2336f809e2dd7d2912c61c8ef02c26')
key = split_target(target, 'backup.npy')
key_2 = get_collision(key)

assert hash(key) ==
bytes.fromhex('61b5649e894a15a053276c0dc828ee64ec2336f809e2dd7d2912c61c8ef02c26')
assert hash(key_2) ==
bytes.fromhex('61b5649e894a15a053276c0dc828ee64ec2336f809e2dd7d2912c61c8ef02c26')
print("INFO: Key 1 as hex", key.hex())
print("INFO: Key 2 as hex", key_2.hex())
print()
print("INFO: Key 1 as plain", key)
print("INFO: Key 2 as plain", key_2)
```