Crypto Funhouse

Crypto Funhouse

Learn to use and break a variety of ciphers and try some CTF-style challenges

[Task 1] It looks secure, so it must be secure - Easy

This room contains a variety of crypto challenges, from a steady intro at the start with some trickier CTF-style challenges at the end.
If you've done this before, you might want to skip to a later section.
Encodings such as hex and base64 are commonly used to encode data, converting the data into a more convinient form.
To the untrained eye, data such as 68656c6c6f and d29ybGQ= looks like gobbledegook. Unfortunately, to those who know

what they're doing it is trivial to decode.
The first code is called hex and the second is base64. Knowing how to decode both is an essential part of your CTF toolkit.
There are many websites available to decode this, or you can use a scripting language to do it. The binascii library of Python is good!

#1
Decode the text 7472796861636b6d65. What does it say

```
import binascii

# Decode as hex
input = "7472796861636b6d65"
output = binascii.unhexlify(input)
# Convert to string and remove b' and '
output = str(output)[2:-1]
print(output)
```

hex

tryhackme

#2
We've written this text in a different form. Can you still read what it says? \x6d\x65\x68\x61\x63\x6b\x74\x72\x79

```
import binascii

# Decode as hex
input = "7472796861636b6d65"
output = binascii.unhexlify(input)
# Convert to string and remove b' and '
output = str(output)[2:-1]
print(output)
```

hex

mehacktry

#3
How about the text YmFzZTY0?

```
import base64

# Decode as base64
input = "YmFzZTY0"
output = base64.b64decode(input)
# Convert to string and remove b' and '
output = str(output)[2:-1]
print(output)
```

base64

base64

#4

Under the HTTP standard, credentials for basic authentication are genuinely 'encrypted' using base64 encoding. Not much good for your bank details! Try to decode the following credentials and give the password:
dXNlcjEyMzpzdHIwbmdQNCQkV29yZA==

```
import base64

# Decode as base64
input = "dXNlcjEyMzpzdHIwbmdQNCQkV29yZA=="
creds = base64.b64decode(input)
# Convert to string and remove b' and '
creds = str(creds)[2:-1]
# Split around the ':' and print the second item
password = creds.split(':')[1]
print(password)
```

base64

user123:str0ngP4$$Word

str0ngP4$$Word

[Task 2] Simple Ciphers - Medium

**One of the simplest ciphers available is the shift cipher - also known as caesar or rot13. Here, each letter is rotated by
a certain number of letters in the alphabet. So, the plaintext "julius" with a shift of 3 goes to the ciphertext "MXOLXV".
To decrypt each letter, move backwards 3 positions in the alphabet. So *M* goes back to *j*, *X* goes back to *u* etc. Have a
look at the following alphabet if your confused:
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
This only allows us to encrypt text. The xor cipher takes this idea and applies it to arbitrary binary data, which we can
store as hex. For each byte of the input, we take the exclusive or of this byte with a secret key. For example, the following
bytes 786f72 representing the text "xor" can be encrypted with the key 0xFF to 87908d. To decrypt, apply the xor operation
on each byte and then decode as hex.**

**The shift and xor ciphers both have a small number of keys and can easily be brute forced (trying every possible key
and looking at the output). The subsitution cipher prevents this attack, by assigning a random permutation of the alphabet
with which to substitute letters. For example:**

Plaintext alphabet: a b c d e f g h i j k l m n o p q r s t u v w x y z

Substitution alphabet: X H W Y M N P R F J O A C Z U L G K T B I V S Q D E

For example, *a* encrypts to *X*, *b* encrypts to *H* and so on.

**There are 26! = 403291461126605635584000000 possible permutations of the alphabet. That's a lot, and we can't brute
force this. However, a new attack called frequency analysis is possible. In english, the letter E appears much more often
than the letter C which appears much more often by the letter Z. So, by counting the number of times each letter appears,
we can guess which ciphertext letter corresponds to which plaintext letter of the alphabet, and deduce the permutation.**

**A table of frequencies can be found at http://pi.math.cornell.edu/~mec/2003-2004/cryptography/subs/frequencies.html. Note
that this method isn't perfect and requires some trial and error - in a given sample of text R might be more common**

than S,
even though on average in English S is more common. In addition it's possible to write text that intentionally breaks the
frequencies that we would expect. For example, the book Euonia by Christian Bök uses only one vowel per chapter and
writes in vaguely coherent English.

- -

#1
Decrypt the following shift cipher with a shift of 13: PELCGBVFSHA.

```
import string

# Get the alphabet from the string module
alpha = string.ascii_uppercase

input = "PELCGBVFSHA"

output = ""
for letter in input:
    # Get the index of the letter in the alphabet
    index = alpha.index(letter)
    # Subtract 13, modulo 26 and get new letter
    new = alpha[(index - 13) % 26]
    output += new

print(output)
```

ROT13

CRYPTOISFUN

- -

#2
Decrypt the following xor cipher with a key byte of 0xAB: 8fd2c6939bc7d8f49fd998f4fb9b9e9e9a93c798f4df9b9b8a8a8a

```
import binascii

input = "8fd2c6939bc7d8f49fd998f4fb9b9e9e9a93c798f4df9b9b8a8a8a"

# Decode input from hex string to bytes
bytes = binascii.unhexlify(input)

out = ""
for byte in bytes:
    # xor with key byte
    dec = byte ^ 0xAB
    # Convert to character
    out += chr(dec)

print(out)
```

XOR

$ym80ls_4r3_P05518l3_t00!!!

- -

#3
Since shift ciphers have only 25 possible keys, and xor ciphers only 255, either can be easily brute forced. Have a go at brute forcing the xor cipher with an unknown key byte: feedfaf1d7fbedebfdfaed

```
import binascii

input = "feedfaf1d7fbedebfdfaed"

# Decode input from hex string to bytes
bytes = binascii.unhexlify(input)

for key in range(0, 0x100):
    out = ""
    for byte in bytes:
        # xor with key byte
        dec = byte ^ key
        # Convert to character
        out += chr(dec)

    print("Key " + hex(key) + ": " + out)
```

XOR Brute
Key 0x88: very_secure

very_secure

#4

Use frequency analysis to decode the following text and give the name of the author:
SMJ YEOYVLUO UOOUWRJGJWS ZI SMJ UKJD ZI SMJ XEIIJOJWYJ JWREWJ OZVWX LUORJ
YJWSOUL HMJJLD LJX SZ SMJ GZDS JKSJWXJX POZDPJYSD. SMJ HMZLJ ZI UOESMGJSEY
WZH UPPJUOJX HESMEW SMJ ROUDP ZI GJYMUWEDG. U CURVJ RLEGPDJ JCJW ZI UW
UWULTSEYUL JWREWJ US LJWRSM ZPJWJX ZVS, UWX E PVODVJX HESM JWSMVDEUDG
SMJ DMUXZHT CEDEZW. SMJ XOUHEWRD UWX SMJ JKPJOEGJWSD HJOJ ZI SMJ GZDS
YZDSLT FEWX. XOUISDGJW ZI SMJ MERMJDS ZOXJO HJOJ WJYJDDUOT SZ JYZWZGEQJ
SMJ LUAZVO ZI GT ZHW MJUX; HMELDS DFELLJX HZOFGJW HJOJ OJNVEOJX SZ JKJYVSJ
SMJ JKPJOEGJWSUL GUYMEWJOT SZ HMEYM E HUD ZALERJX YZWDSUWSLT SZ MUCJ
OJYZVODJ.
EW ZOXJO SZ YUOOT ZVS GT PVODVESD DVYYJDDIVLLT, E MUX PVOYMUDJX U MZVDJ
HESM UAZCJ U NVUOSJO ZI UW UYOJ ZI ROZVWX EW U CJOT NVEJS LZYULEST. GT
YZUYM-MZVDJ HUD WZH YZWCJOSJX EWSZ U IZORJ UWX U IZVWXOT, HMELDS GT
DSUALJD HJOJ SOUWDIZOGJX EWSZ U HZOFDMZP. E AVELS ZSMJO JKSJWDECJ
HZOFDMZPD GTDJLI, UWX MUX U IEOJ-POZZI AVELXEWR IZO GT XOUHEWRD UWX
XOUISDGJW. MUCEWR GTDJLI HZOFJX HESM U CUOEJST ZI SZZLD, UWX MUCEWR
DSVXEJX SMJ UOS ZI YZWDSOVYSEWR JUYM ZI SMJG, E US LJWRSM LUEX ES XZHW
UD U POEWYEPLJ—SMUS, JKYJPS EW OUOJ YUDJD, E HZVLX WJCJO XZ UWTSMEWR
GTDJLI EI E YZVLX UIIZOX SZ MEOJ UWZSMJO PJODZW HMZ YZVLX XZ ES IZO GJ.

frequency analysis and a quick Google Search for the author

Charles Babbage

[Task 3] Advanced Encryption - Medium

**The Advanced Encryption Standard, AES, also known as Rijndael, is one of the most trusted algorithms for modern encryption. It is a
block cipher, meaning it operates on blocks of 16 bytes at once. When used correctly, it cannot be broken by modern computational
technology. For each block, it applies the following operations to a 16-byte state represented as a 4x4 matrix. This process is iterated
over several rounds:**

- **SubBytes: Uses a lookup table to substitute each byte with some other byte**
- **ShiftRows: Shifts each row of the 4x4 matrix, swapping the positions of bytes in the 16-byte state**
- **MixColumns: Combines and mixes the contents of each column in the 4x4 matrix**
- **AddRoundKey: Xors the current state with a 16-byte round key derived from the main key used**
-

You can find full details of each step at https://en.wikipedia.org/wiki/Advanced_Encryption_Standard.

In the final round, the MixColumns step is skipped. In addition, an additional AddRoundKey step is applied using the master key before the first round.

Many rounds of Rijndael are applied to ensure secure encryption. A master key of 128, 192 or 256 bits is chosen, and 128-bit round keys are
derived for each round. 10 rounds are used in AES-128, 12 for AES-192 and 14 for AES-256. The process for deriving the round keys from the
master key is described at https://en.wikipedia.org/wiki/Rijndael_key_schedule.

The SubBytes step substitutes each byte of the state according to a hardcoded lookup table called the s-box. For example, the byte 0x00 is substituted
for the byte 0x63. The full lookup table can be found at https://en.wikipedia.org/wiki/Rijndael_S-box.

The ShiftRows operation takes the current state as a 4x4 matrix:

```
   x0   x4   x8   x12
   x1   x5   x9   x13
   x2   x6   x10  x14
   x3   x7   x11  x15
```

It then rotates each row by a different number of positions:

```
   x0   x4   x8   x12    (no shift)
   x5   x9   x13  x1     (shift 1 left)
   x10  x14  x2   x6     (shift 2 left)
   x15  x3   x7   x11    (shift 3 left)
```

The MixColumns step Mixes the bytes of every column of the state, according to a linear transformation. Operations are done in the field GF(2^8),
meaning addition and multiplication works a little differently - this detail isn't too important.
Finally, the AddRoundKey step is simply an XOR (exclusive or) operation. This means that every byte of the state is xored with the corresponding
byte of the round key.

#1

Given the state [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15], apply the ShiftRows operation as described above and give the output in the same notation

```python
def ShiftRows(state):
    new = [-1]*16
    # First row, no shift
    new[0] = state[0]
    new[4] = state[4]
    new[8] = state[8]
    new[12] = state[12]
    # Second row, shift left 1
    new[1] = state[5]
    new[5] = state[9]
    new[9] = state[13]
    new[13] = state[1]
    # Third row, shift left 2
    new[2] = state[10]
    new[6] = state[14]
    new[10] = state[2]
    new[14] = state[6]
    # Fourth row, shift left 3
    new[3] = state[15]
    new[7] = state[3]
    new[11] = state[7]
    new[15] = state[11]
    return new

input = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
output = ShiftRows(input)
print(output)
```

ShiftRows

0, 5, 10, 15, 4, 9, 14, 3, 8, 13, 2, 7, 12, 1, 6, 11

- -

#2
What is the output of applying the SubBytes step to the byte 0x75? Give answer as lower-case hex, e.g. 0x1b.

```
sbox = [0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5,
        0x30, 0x1, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
        0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0,
        0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,
        0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc,
        0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
        0x4, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x5, 0x9a,
        0x7, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,
        0x9, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0,
        0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,
        0x53, 0xd1, 0x0, 0xed, 0x20, 0xfc, 0xb1, 0x5b,
        0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
        0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85,
        0x45, 0xf9, 0x2, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,
        0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5,
        0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,
        0xcd, 0xc, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17,
        0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
        0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88,
        0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0xb, 0xdb,
        0xe0, 0x32, 0x3a, 0xa, 0x49, 0x6, 0x24, 0x5c,
        0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
        0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9,
        0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x8,
        0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6,
        0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
        0x70, 0x3e, 0xb5, 0x66, 0x48, 0x3, 0xf6, 0xe,
        0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,
        0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94,
        0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
        0x8c, 0xa1, 0x89, 0xd, 0xbf, 0xe6, 0x42, 0x68,
        0x41, 0x99, 0x2d, 0xf, 0xb0, 0x54, 0xbb, 0x16]

input = 0x75
output = sbox[input]
print(hex(output))
```

SubBytes

0x9d

[Task 4] CTF style challenge - Hard

A new method of AES encryption, AES-ONE is invented that uses only a single round of the Rijndael cipher. A sample set of ciphertexts,
all produced from english text, is provided in the file ciphertexts.txt. No punctuation is included: just lower-case letters and spaces.
For example, a sample plaintext could be "tricky ctf chall" or 747269636b792063746620636861c6c. Every ciphertext in the file uses the
same 128-bit master key.

These tasks are quite tricky - use the hints or look at the writeup if you're stuck!

If you need help, the plaintext for the sample ciphertexts can be found here: **http://-historyofeconomicthought.mcmaster.ca/babbage/index.html*.***
Try to solve the problem without it first!

Download File

#1

Can you break the ciphertext 67a2401f0f36c3b680abb775ecedf311, encrypted with the same key as the sample ciphertexts? (Give answer as text, e.g. "solved ctf chall", without quotes)

You don't need to fully break the cipher. Just break it enough to figure out what the target ciphertext says!

2 py files and 1 txt used to solve

rijndael is king

#2

What technique, covered earlier in this room, was used to break the above ciphertext?

frequency analysis

#3

Bonus challenge: what key was used to encrypt these ciphertexts? (Give answer in hex, e.g. 112233445566778899aabbccddeeff. You will need to learn about the Rijndael key schedule)

```python
import binascii

# Standard AES s-box
sbox = [0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x1, 0x67, 0x2b, 0xfe, 0xd7, 0xab,
0x76, 0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72,
0xc0, 0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31,
0x15, 0x4, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x5, 0x9a, 0x7, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2,
0x75, 0x9, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f,
0x84, 0x53, 0xd1, 0x0, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58,
0xcf, 0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x2, 0x7f, 0x50, 0x3c, 0x9f,
0xa8, 0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3,
0xd2, 0xcd, 0xc, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19,
0x73, 0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0xb,
0xdb, 0xe0, 0x32, 0x3a, 0xa, 0x49, 0x6, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4,
0x79, 0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae,
0x8, 0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b,
0x8a, 0x70, 0x3e, 0xb5, 0x66, 0x48, 0x3, 0xf6, 0xe, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d,
0x9e, 0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28,
0xdf, 0x8c, 0xa1, 0x89, 0xd, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0xf, 0xb0, 0x54, 0xbb,
0x16]

# Inverse shift rows to deduce key positions
invShiftRows = [0, 5, 10, 15, 4, 9, 14, 3, 8, 13, 2, 7, 12, 1, 6, 11]

# ShiftRows function for the plaintexts
def ShiftRows(state):
    new = [-1]*16
    # First row, no shift
    new[0] = state[0]
    new[4] = state[4]
    new[8] = state[8]
    new[12] = state[12]
    # Second row, shift left 1
    new[1] = state[5]
    new[5] = state[9]
    new[9] = state[13]
    new[13] = state[1]
    # Third row, shift left 2
    new[2] = state[10]
    new[6] = state[14]
    new[10] = state[2]
    new[14] = state[6]
    # Fourth row, shift left 3
    new[3] = state[15]
    new[7] = state[3]
    new[11] = state[7]
    new[15] = state[11]
    return bytes(new)

def getKey(number, posssibles):
    # Takes a number between 0 and 2^16 - 1 and returns a unique key
    key = [-1]*16
    for i in range(0, 16):
        key[i] = possibles[i][number % 2]
        number >>= 1
    return bytes(key)

def getRoundKey(key):
    # Given an 16-byte master key, get the first round key
    roundKey = [-1]*16
    # First column (W4)
    roundKey[0] = key[0] ^ sbox[key[13]] ^ 1
    roundKey[1] = key[1] ^ sbox[key[14]]
    roundKey[2] = key[2] ^ sbox[key[15]]
    roundKey[3] = key[3] ^ sbox[key[12]]
    # Second column (W5)
    roundKey[4] = key[4] ^ roundKey[0]
    roundKey[5] = key[5] ^ roundKey[1]
    roundKey[6] = key[6] ^ roundKey[2]
    roundKey[7] = key[7] ^ roundKey[3]
    # Third column (W6)
    roundKey[8]  = key[8]  ^ roundKey[4]
    roundKey[9]  = key[9]  ^ roundKey[5]
    roundKey[10] = key[10] ^ roundKey[6]
    roundKey[11] = key[11] ^ roundKey[7]
    # Fourth column (W7)
    roundKey[12] = key[12] ^ roundKey[8]
    roundKey[13] = key[13] ^ roundKey[9]
    roundKey[14] = key[14] ^ roundKey[10]
    roundKey[15] = key[15] ^ roundKey[11]
    return bytes(roundKey)

def encrypt(input, key):
    # Encrypts a 16-byte input with the 16-byte master key provided
    roundKey = getRoundKey(key)
    key = ShiftRows(key)
    state = list(input)
    for i in range(0, 16):
        # AddRoundKey with master key
        state[i] ^= key[i]
        # SubBytes
        state[i] = sbox[state[i]]
        # AddRoundKey with first round key
        state[i] ^= roundKey[i]
    return bytes(state)

# Load in the two plaintexts and apply ShiftRows
p1 = ShiftRows(b"the economy of m")
p2 = ShiftRows(b"achinery and man")
# Load in the two ciphertexts
c1 = binascii.unhexlify("0c9e246cb8a1bfa3b0e947a1a94c8d11")
c2 = binascii.unhexlify("998fffc15bd9cf74804d59fc23cb6e59")

# Find key values for each position
possibles = [None]*16
for i in range(0, 16):
    # Calculate E(x) ^ E(x')
    expected = c1[i] ^ c2[i]
    # Brute force values of s(x ^ k) ^ s(x' ^ k)
    for k in range(0, 256):
        result = sbox[p1[i] ^ k] ^ \
                 sbox[p2[i] ^ k]
        if result == expected:
            # Find real position of key by reversing ShiftRows
            pos = invShiftRows[i]
            print("Key byte for pos " + str(pos) + ":", hex(k), chr(k))
            # Store the possible key value
            if possibles[pos] == None:
                possibles[pos] = k
            else:
                possibles[pos] = (possibles[pos], k)

# Brute force the 2^16 possibilities
print("")
print("Searching keys...")
print("=================")
for i in range(0, 2**16):
    key = getKey(i, possibles)
    if c1 == encrypt(p1, key) and c2 == encrypt(p2, key):
        print("Key is", key.hex(), "(" + str(key) + ")")
```

```
Key is 4e30745f73302d346456616e43336421 (b'N0t_s0-4dVanC3d!')
```

4e30745f73302d346456616e43336421

discussion

CryptoFunHouse CTF-style challenges

Both of these challenges are rather difficult. Don't be disheartened if you have to use the sample plaintext or part of this writeup to solve them.

Question 1

Usually, the Rijndael cipher (AES) is iterated over several rounds to ensure a secure encryption: for AES with a 128-bit key, 10 rounds are necessary. It comes as no surprise that when only a single round is used, the cipher is very weak.
The key point is that in Rijndael, the MixColumns step it not applied on the final round. With only one round, the MixColumns operation never happens. We must also remember that an additional AddRoundKey operation is applied with the master key before the rest of the algorithm. As such, the encryption process for AES-ONE looks like this:

- **AddRoundKey with master key**
- **SubBytes**
- **ShiftRows**
- **AddRoundKey with first round key**

If you've forgotten how any of these three operations work, have a quick read here.

Notice in the SubBytes operation, the input value of one byte of the state has no impact on the output value of any other byte. In addition, the position of that byte in the state has no impact on its output value. As such, the ShiftRows and SubBytes operations can be swapped (in maths terms, we say the operations commute). Therefore, the encryption process of AES-ONE is equivalent to:

- **AddRoundKey with master key**
- **ShiftRows**
- **SubBytes**
- **AddRoundKey with first round key**

Now, in the AddRoundKey step, once again, the value of any one byte has no impact on the value of any other byte during the operation (remember, each byte is xored with the corresponding byte of the key). This time, the byte's position in the array does have an impact, as each byte of the key will usually be different, so the ShiftRows operation is not commutative with AddRoundKey. However, we can swap the order of AddRoundKey and ShiftRows if we first apply ShiftRows to the master key. Thus, our encryption process is equivalent to:

- **ShiftRows**
- **AddRoundKey with ShiftRows(master key)**
- **SubBytes**
- **AddRoundKey with first round key**

Since both the keys are fixed throughout all the ciphertexts in the challenge, we can model the AddRoundKey, SubBytes and AddRoundKey steps as some unknown permutation that maps every byte onto some other byte. This permutation will be different for each of the 16 positions of the state. The encryption process is now fairly simple:

- **ShiftRows**
- **Unknown permutation**

The cipher now looks vaguely similar. Remember how the substitution cipher, covered in part 2, is essentially a permutation of the alphabet, where each letter of the alphabet is mapped to another letter of the alphabet:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

X H W Y M N P R F J O A C Z U L G K T B I V S Q D E

So A->X, B->H etc. Our new unknown permutation is fairly similar: each byte between 0 and 255 is mapped to some byte between 0 and 255. In addition, we know that the plaintext is lower-case english text with spaces. So we would expect to see a total of 27 different bytes in each position, when we analyse the ciphertexts. We can then perform frequency analysis, with a bit of manual guesswork, to deduce the mapping of bytes in the ciphertext to english letters. Crucially, and perhaps most challengingly, there is a different mapping for each of the 16 positions in the AES state - so we need to perform 16 different sets of frequency analysis to deduce the solution, and the online substitution cipher solvers will not work.
This is rather difficult, since frequency analysis isn't perfect and some letters will need swapping for each of the 16 positions. To help with this, the target plaintext contains the word "rijndael", in the hopes of providing a recognizable word to use as an anchor point when swapping letters and cracking the cipher. By swapping letters and developing a mapping for each of the 16 positions so that the ciphertexts roughly decode as English, it is possible to break the cipher and discover that the target plaintext is "rijndael is king". A sample solution is provided in question_1.py, which uses the manually developed frequency orderings in freq.py. This doesn't decode the entire plaintext, but the first few plaintexts and the target all decode in their entirety. With more work, or a smarter algorithm, the entire plaintext could be decoded.
A slightly easier, but still impressive solution was to use the provided plaintext to break the cipher. Spaces are reliably the most common 'letter' in English, so it is easy to deduce that the start of the plaintext is something of the form "xxx xxxxxxx xx xxxxxxxxx xxx xxx". Looking at the linked plaintext, we know this is "the economy of

machinery and man". By comparing the plaintext to the bytes of the ciphertext with the bytes of the plaintext (remembering that everything is lower case and there is no punctuation), we can deduce the 16 permutations of bytes to letters without needing full frequency analysis. In the crypto world we'd call this a known plaintext attack since knowing a sample of plaintext-ciphertext pairs, we can go on to break other ciphertexts. The provided plaintext can also be used with the first solution to fine-tune the mappings given by frequency analysis, during the phase of manually swapping letters.

Question 2

Given two plaintext-ciphertext pairs, it is possible to deduce the key used in the encryption, assuming a 128-bit master key is used. Let's take two plaintext-ciphertext pairs deduced through our frequency analysis in the first part:

"the economy of m": 7468652065636f6e6f6d79206f66206d -> 0c9e246cb8a1bfa3b0e947a1a94c8d11

"achinery and man": 616368696e65727920616e64206d616e -> 998fffc15bd9cf74804d59fc23cb6e59

Remember that the AES-ONE encryption process is equivalent to ShiftRows followed by our unknown permutation of AddRoundKey and SubBytes. To makes things simple, let's apply ShiftRows to our plaintexts right away:

"tcymem ofenoho ": 7463796d656d20206f66656e6f686f20 -> 0c9e246cb8a1bfa3b0e947a1a94c8d11

"aennnaai mhy crd": 61656e6e6e616169206d687920637264 -> 998fffc15bd9cf74804d59fc23cb6e59

Hence the encryption operation is now:

- AddRoundKey with ShiftRows(master key)
- SubBytes
- AddRoundKey with first round key
-

Let's get some notation. Let x be a byte of the plaintext, and r be the corresponding byte of the first round key, and k be the corresponding byte of the 128-bit master key after ShiftRows. We'll also say that E(x) represents the encrypted output of byte x, and s(x) represents the output of byte x in the AES s-box. Putting this all together, we can create an equation for the encryption process of AES-ONE:
$E(x) = s(x \oplus k) \oplus r$
Take byte x', where x' is from a different plaintext and is in the same position as byte x. We can now write an equation for the encryption of x':
$E(x') = s(x' \oplus k) \oplus r$
Now, let's combine the two equations and cancel out the round key r:
$E(x) \oplus E(x') = s(x \oplus k) \oplus s(x' \oplus k) \oplus r \oplus r$
$E(x) \oplus E(x') = s(x \oplus k) \oplus s(x' \oplus k)$
The values of E(x), E(x'), x and x' are known through our two plaintext-ciphertext pairs. The unknown key value k appears in both instances of the sbox, meaning we can't rearrange the equation in terms of k like we did back in school. However, there are only 256 possible values of the key, meaning we can brute force the value of k and find the value which satisfies the equation.
Unfortunately, when we try this, we don't get a unique value for each key byte. In fact, we end up with two possible values for each position (see code in question_2.py):

```
Key byte for pos 0: 0x4e N
Key byte for pos 0: 0x5b [
Key byte for pos 5: 0x30 0
Key byte for pos 5: 0x36 6
Key byte for pos 10: 0x61 a
Key byte for pos 10: 0x76 v
Key byte for pos 15: 0x21 !
Key byte for pos 15: 0x22 "
Key byte for pos 4: 0x73 s
Key byte for pos 4: 0x78 x
Key byte for pos 9: 0x56 V
Key byte for pos 9: 0x5a Z
Key byte for pos 14: 0x25 %
Key byte for pos 14: 0x64 d
Key byte for pos 3: 0x16
Key byte for pos 3: 0x5f _
Key byte for pos 8: 0x2b +
Key byte for pos 8: 0x64 d
Key byte for pos 13: 0x33 3
Key byte for pos 13: 0x38 8
Key byte for pos 2: 0x74 t
Key byte for pos 2: 0x79 y
Key byte for pos 7: 0x23 #
Key byte for pos 7: 0x34 4
Key byte for pos 12: 0xc
Key byte for pos 12: 0x43 C
Key byte for pos 1: 0x30 0
Key byte for pos 1: 0x3b ;
Key byte for pos 6: 0x2d -
Key byte for pos 6: 0x30 0
Key byte for pos 11: 0x2a *
Key byte for pos 11: 0x6e n
```

If you got this far, well done. To solve for a unique key value we'll need to go back to our original equations. Where Xi is the input byte in the ith position, Ki is the key byte in the ith position after ShiftRows and Ri is the first round key byte in the ith position, we can say that:
E(Xi) = s(Xi ⊕ Ki) ⊕ Ri, and
E(Xi') = S(Xi' ⊕ Ki) ⊕ Ri, for 0 <= i < 16.
Thus we have 32 equations that the master key must satisfy. Since there are two possibilities for each of the 16 positions, there are 16 possible keys to check. Through the Rijndael key schedule, the first round key is derived from the master key (although each byte of the round key can depend on several bytes of the master key, so we must now brute force the entire 16-byte master key, rather than individual bytes). The values of the round keys follow a recurrence relationship starting from the master key, implemented in question_2.py as the function getRoundKey. The Rijndael key schedule is described in detail here.
The Python solution script can easily brute force these possibilities and deduce that the key is 4e30745f73302d346456616e43336421, or "N0t_s0-4dVanC3d!" as text.

More Info

If you're interested, the program used to create the challenge can be found under the go_program directory. Assuming you have Go installed, you can run it with:

```
go get https://github.com/xrmon/aes
go run one.go
```

This will download & run the AES implementation at https://github.com/xrmon/aes.
The book used in the challenge was The Economy of Machinery and Manufactures, by Charles Babbage (available in three parts here). To make the challenge easier all punctuation and numbers were removed, roman numerals were converted to text, and letters were converted to lower case. The book is otherwise unaltered.