superkojiman Change extension to .md `6fb5ce1` on Feb 9, 2016

1 contributor

381 lines (274 sloc) 12.9 KB

Solved by superkojiman

Block is a 130 point cryptography challenge. I should mention that crypto isn't my forte so my solution is probably not the most efficient. But hey, it worked.

> Daedalus Corp has been using this script to encrypt it's data! We think this file contains a password to their command server. Can you crack it?

We're given two files; a file called encrypted, which contains the flag, and a python script block.py, which allows us to encrypt and decrypt files.

Here's what block.py looks like

```
#!/usr/bin/python2
from sys import argv, exit
import struct

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

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

def S(block, SBoxes):
    output = 0
    for i in xrange(0, len(SBoxes)):
        output |= SBoxes[i][(block >> 4*i) & 0b1111] << 4*i

    return output


PBox = [13, 3, 15, 23, 6, 5, 22, 21, 19, 1, 18, 17, 20, 10, 7, 8, 12, 2, 16, 9, 14, 0, 11, 4]
PInvBox = [21, 9, 17, 1, 23, 5, 4, 14, 15, 19, 13, 22, 16, 0, 20, 2, 18, 11, 10, 8, 12, 7, 6, 3]
def permute(block, pbox):
    output = 0
    for i in xrange(24):
        bit = (block >> pbox[i]) & 1
        output |= (bit << i)
    return output

def encrypt_data(data, key):
    enc = ""
    for i in xrange(0, len(data), 3):
        block = int(data[i:i+3].encode('hex'), 16)

        for j in xrange(0, 3):
            block ^= key
            block = S(block, SBoxes)
            block = permute(block, PBox)

        block ^= key

        enc += ("%06x" % block).decode('hex')

    return enc

def decrypt_data(data, key):
    dec = ""
    for i in xrange(0, len(data), 3):
        block = int(data[i:i+3].encode('hex'), 16)

        block ^= key
```

```
        for j in xrange(0, 3):
            block = permute(block, PInvBox)
            block = S(block, SInvBoxes)
            block ^= key

        dec += ("%06x" % block).decode('hex')

    return dec

def encrypt(data, key1, key2):
    encrypted = encrypt_data(data, key1)
    encrypted = encrypt_data(encrypted, key2)
    return encrypted

def decrypt(data, key1, key2):
    decrypted = decrypt_data(data, key2)
    decrypted = decrypt_data(decrypted, key1)
    return decrypted

def usage():
    print "Usage: %s [encrypt/decrypt] [key1] [key2] [in_file] [out_file]" % argv[0]
    exit(1)

def main():
    if len(argv) != 6:
        usage()

    if len(argv[2]) > 6:
        print "key1 is too large"
    elif len(argv[3]) > 6:
        print "key2 is too large"

    key1 = int(argv[2], 16)
    key2 = int(argv[3], 16)

    in_file = open(argv[4], "r")

    data = ""
    while True:
        read = in_file.read(1024)
        if len(read) == 0:
            break

        data += read

    in_file.close()

    if argv[1] == "encrypt":
        data = "message: " + data
        if len(data) % 3 != 0: #pad
            data += ("\x00" * (3 - (len(data) % 3)))

        output = encrypt(data, key1, key2)
    elif argv[1] == "decrypt":
        output = decrypt(data, key1, key2)
    else:
        usage()


    out_file = open(argv[5], "w")
    out_file.write(output)
    out_file.close()

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

If we run block.py, we get a helpful usage message:

```
$ ./block.py
Usage: ./block.py [encrypt/decrypt] [key1] [key2] [in_file] [out_file]
```

The script does double encryption/decryption and requires two keys. The first key is used for the first encryption/decryption layer, and the second key is used for the second layer. If we examine the code, we see that the key needs to be at most, 6 characters in length.

```
if len(argv[2]) > 6:
    print "key1 is too large"
elif len(argv[3]) > 6:
    print "key2 is too large"
```

Each key is then converted into it's hexadecimal value, which means the range of allowed characters is from 0-9 and A-F.

```
key1 = int(argv[2], 16)
key2 = int(argv[3], 16)
```

Therefore the allowed keys fall in the range of 0x000000 to 0xFFFFFF, which means there are a total of 16,777,215 possible keys. That's not too bad.

Let's take a look at the encryption phase first. The script reads the contents of the input file into a variable called data. It then prefixes data with a string "messages: ", and then encrypts the entire thing. I'll refer to the string "messages: " as the header from here on.

```
if argv[1] == "encrypt":
    data = "message: " + data
    if len(data) % 3 != 0: #pad
        data += ("\x00" * (3 - (len(data) % 3)))
    output = encrypt(data, key1, key2)
```

If we look at the encrypt() function, we see that it does the double encryption:

```
def encrypt(data, key1, key2):
    encrypted = encrypt_data(data, key1)
    encrypted = encrypt_data(encrypted, key2)
    return encrypted
```

Now because it prefixes data with a known header, we can execute a known-plaintext attack. Let's see it in action. First, modify the script so that it prints out both encrypted values:

```
def encrypt(data, key1, key2):
    encrypted = encrypt_data(data, key1)
    print "encryption layer 1:", encrypted.encode("hex")

    encrypted = encrypt_data(encrypted, key2)
    print "encryption layer 2:", encrypted.encode("hex")

    return encrypted
```

Now we'll create a file to encrypt and run the script to see what happens.

```
$ echo "abcd" > m1.txt
$ ./block.py encrypt 000001 000002 in.txt out.txt
encryption layer 1: be2369c422c0017f979b23b7dcd840
encryption layer 2: f90ce5fac97c7d18ae74ce69beb0bf
```

After the first layer of encryption is applied, our plaintext maps to the ciphertext like so:

```
be 23 69 c4 22 c0 01 7f 97 9b 23 b7 dc d8 40
m  e  s  s  a  g  e  s  :     a  b  c  d  \n
```

Since we have a small key space, it's feasible to encrypt the header using all possible keys and store the encrypted header and its corresponding key in a hashmap for a quick lookup:

```
encrypted_header: key
```

So now we have a hashmap of all possible ciphertext for the header, encrypted with all possible keys. In our example, we know the encrypted header is be2369c422c0017f97 after the first layer of encryption is applied. After the second layer of encryption is applied, the data becomes f90ce5fac97c7d18ae74ce69beb0bf.

To find key2, all we need to do is use all possible keys to remove the second layer of encryption from f90ce5fac97c7d18ae74ce69beb0bf and grab the first 9 bytes, which is the encrypted header. We then lookup this encrypted header from our hashmap, and if we find it, then we have both key1, and key2 that was used to encrypt the file.

To summarize:

1. Create a hashmap that contains the header encrypted with all possible keys. So we would have something like this: "be2369c422c0017f979": "000001"
2. For all possible keys, remove the second encryption layer and get the first 9 bytes of the decrypted string (string encrypted with first layer), which is the encrypted header.
3. Lookup the encrypted header in our hashmap. If we find it, then we have key2, which was used to remove the second layer of encryption, and key1, which is the value associated with the encrypted header in our hashmap.

Here's the final script:

```
#!/opt/local/bin/python2

import struct, sys

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

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

def S(block, SBoxes):
    output = 0
    for i in xrange(0, len(SBoxes)):
        output |= SBoxes[i][(block >> 4*i) & 0b1111] << 4*i

    return output


PBox = [13, 3, 15, 23, 6, 5, 22, 21, 19, 1, 18, 17, 20, 10, 7, 8, 12, 2, 16, 9, 14, 0, 11, 4]
PInvBox = [21, 9, 17, 1, 23, 5, 4, 14, 15, 19, 13, 22, 16, 0, 20, 2, 18, 11, 10, 8, 12, 7, 6, 3]
def permute(block, pbox):
    output = 0
    for i in xrange(24):
        bit = (block >> pbox[i]) & 1
        output |= (bit << i)
    return output

def encrypt_data(data, key):
    enc = ""
    for i in xrange(0, len(data), 3):
        block = int(data[i:i+3].encode('hex'), 16)

        for j in xrange(0, 3):
            block ^= key
            block = S(block, SBoxes)
            block = permute(block, PBox)

        block ^= key

        enc += ("%06x" % block).decode('hex')

    return enc

def decrypt_data(data, key):
    dec = ""
    for i in xrange(0, len(data), 3):
        block = int(data[i:i+3].encode('hex'), 16)

        block ^= key
        for j in xrange(0, 3):
            block = permute(block, PInvBox)
            block = S(block, SInvBoxes)
            block ^= key

        dec += ("%06x" % block).decode('hex')
```

```
    return dec

def encrypt(data, key1):
    encrypted = encrypt_data(data, key1)
    return encrypted

def decrypt(data, key2):
    decrypted = decrypt_data(data, key2)
    return decrypted

def main():

    keymap = {}

    status = 0
    for i in range(0x000000, 0xFFFFFF):
        key1 = i

        # build a hashmap of "message: " encrypted with the first layer of encryption
        # using all keys from 0x000000 to 0xFFFFFF
        data = "abcd"   # this can be anything, we just want the header
        data = "message: " + data

        if len(data) % 3 != 0: #pad
            data += ("\x00" * (3 - (len(data) % 3)))

        output = encrypt(data, key1)
        header = output.encode("hex")[:18]

        # this might take a while, so print something so we know it's doing stuff
        if status == 10000:
            print "status: %06x of %06x  key1: %d -> %s  header: %s" % (key1, 0xFFFFFF, key1, output.encode
            status = 0

        status += 1
        keymap[header] = key1

    print "keymap size:", len(keymap)

    # now we'll decrypt the second layer of "encrypted" and see if the header matches any of the
    # the ones we generated. if it does, then we have both keys used for the encryption

    # read in contents of encrypted
    in_file = open("encrypted", "r")
    data = ""
    while True:
        read = in_file.read(1024)
        if len(read) == 0:
            break
        data += read
    in_file.close()

    # remove second layer of encryption
    status=0
    for i in range(0x000000, 0xFFFFFF):
        key2 = i
        output = decrypt(data, key2)
        header = output.encode("hex")[:18]

        if status == 10000:
            print "key2: %d -> %s  header: %s" % (key1, output.encode("hex"), header)
            status = 0

        # check if we have this header in keymap
        if header in keymap:
            print "Solved! key1: %06x key2: %06x" % (keymap[header], key2)
            sys.exit(0)

        status += 1


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

Let's run it!

```
$ time ./solveit.py
status: 002710 of ffffff  key1: 10000 -> 5de6c1244f049c8b22cca58459c49b  header: 5de6c1244f049c8b22
status: 004e20 of ffffff  key1: 20000 -> 855567efe669b5e4c0045839e2781a  header: 855567efe669b5e4c0
status: 007530 of ffffff  key1: 30000 -> 0061d4f3850fa46707cda20faf3281  header: 0061d4f3850fa46707
status: 009c40 of ffffff  key1: 40000 -> 0141d19c8cbaf8ed25144bc2e2c2ec  header: 0141d19c8cbaf8ed25
.
.
.
trying 74e1e0 of ffffff   key2: 7660000 ->
a47285f2a2773c036efd351c6cec58f1cc3fbcf67c15caec1e1bb6bbb6c4435c159d59270550b2  header:
a47285f2a2773c036e
trying 7508f0 of ffffff   key2: 7670000 ->
b024a1d804d47c95c45d045f456e47412a2895afda004e1bd9ab8b4b046cac656b725300d6cc57  header:
b024a1d804d47c95c4
trying 753000 of ffffff   key2: 7680000 ->
3358a7d2e95c20e496ea6917f30e1c624fb8c8825f69495f64a6a3f5c668ecd70ed48eac7f8171  header:
3358a7d2e95c20e496
trying 755710 of ffffff   key2: 7690000 ->
1c9a46dd8f160adb175098d80f6ec879ecb3ca98bfab2f154a38e1d366c236c5632a292354fe04  header:
1c9a46dd8f160adb17
Solved! key1: b89567 key2: 756c21

real    160m17.938s
user    160m12.445s
sys 0m4.399s
```

Two coffees, a lunch break, and a meeting later, both keys were found! As I said, not the most efficient way to do it, but it worked. Now that we have both keys, we can decrypt the encrypted file:

```
$ ./block.py decrypt b89567 756c21 encrypted flag.txt
$ cat flag.txt
message: 98acd72dda19cac0ceb93be06d1baa
```

The flag is **98acd72dda19cac0ceb93be06d1baa**