Features Business Explore Marketplace Pricing This repository Search Sign in or Sign up

dqi / ctf_writeup Watch 6 Star 16 Fork 5

Code Issues 1 Pull requests 0 Projects 0 Insights

Branch: master ctf_writeup / 2016 / sharifctf / Create new file Find file History

dqi sharifctf XOR Latest commit 364740a on Dec 18 2016

..

README.md sharifctf XOR 5 months ago

README.md

SharifCTF: XOR

Challenge details

Contest	Challenge	Category	Points
Sharif CTF	XOR	crypto	150

Description

> See the attachment.
> Notice that this is Python 3, and msg is a byte array, not a string.`

Write-up

For this challenge we get the following python code and a 301 byte file with encrypted data.

```
#!/usr/bin/env python3

# Hint: Flag starts with "SharifCTF{", and ends with "}".

from secret import p_small_prime, q_small_prime, r, key, flag
flag = flag * r

def encrypt(msg, p, q, r, key):
  enc = []
  for i in range(len(msg)):
        enc += [(i%p) ^ ((msg[i]**q) % (q**2 - 6*q + 6)) ^ key[r*i % len(key)]]
  return bytes(enc)

with open('enc', 'wb') as f:
  f.write(encrypt(flag, p_small_prime, q_small_prime, r, key))
```

The idea

Looking at the script we see that first the flag is being copied *r* times. Since in this CTF the flags so far have been 43 bytes ('SharifCTF{' + [0-9a-f]{32} + '}') this means *r* is equal to 301/43 = 7.

Analyzing the code we see any byte in the plaintext comes from the following three results:

```
x1 = i^p
x2 = (msg[i]**q) % (q**2 - 6*q + 6)
x3 = key[r*i % len(key)]
```

Since we know the position we know *i*, meaning `x1` is only based on *p*, `x2` is only based on *q* and `x3` is only based on the key.

We also know *p* and *q* are *small* primes.

And we also have some known plaintext to use.

These facts combined allow us to, for some starting keylength, pick some *p,q* pair and generate the key which would encrypt 'S' to the seven encrypted occurences of 'S' in the ciphertext. If there is a valid key (i.e indices of this key have no conflicting solutions) we can then validate this key for the encrypted 'h' in the ciphertext. This way we keep validating the key until we either reach the '{' in the know plaintext or the key becomes invalid. In which case we go on to the next *p,q* pair.

The nice thing of this algorithm is that the keylength does not influence the runtime of one loop though all the *p,q*.

This algorithm finds a *p*, *q* and a key which work for the 'SharifCTF{' part of the plaintext, and then uses it to bruteforce the remaining plaintext char-by-char:

Solution

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import itertools
import string

f = open('./enc', 'rb')
data = f.read()

smallprimes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59,
       61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137,
       139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211,
       223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283,
       293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379,
       383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461,
       463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557, 563,
       569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643,
       647, 653, 659, 661, 673, 677, 683, 691, 701, 709, 719, 727, 733, 739,
       743, 751, 757, 761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 829,
       839, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911, 919, 929, 937,
       941, 947, 953, 967, 971, 977, 983, 991, 997]


def check(v):
   return v[0] ^ v[1] ^ v[2] == v[3]


def get_key(v):
   return v[2], v[0] ^ v[1] ^ v[3]


def valid(v):
   # set(v) == different tuples
   # f for... == different first elements
   return len(set(v)) == len({f for f, g in v})


# For a certain key-length guess
for keyl in range(1, 22):
   print 'keylength: %d' % keyl
   # For a prime-pair candidate
   for p,q in itertools.product(smallprimes, repeat=2):
       s = []
       # Generate a key for this combination
       for c, pt in enumerate('SharifCTF{'):
           # This is known for a given p,q
           ilist = range(0 + c, 301, 43)
           senc = [ord(data[i]) for i in ilist]
           x1 = map(lambda x: x % p, ilist)
           x2 = [(ord(pt)**q) % (q**2 - 6 * q + 6)] * 7
           # calculate a possible key for this p, q pair
           indices = map(lambda x: (x * 7) % keyl, ilist)
           s.extend(map(get_key, zip(x1, x2, indices, senc)))
           # Key still valid?
```

```
            if not valid(s):
                break
            if pt == '{':
                s = list(set(s))
                s.sort(key=lambda x: x[0])
                key = map(lambda x: x[1], s)
                # print 'decrypting with \np: %d\nq: %d\nkey: %s' % (p, q, ''.join(map(lambda x: chr(x), key)))
                # Bruteforce pt char-by-char
                pt = ''
                for i in range(len(data)):  # or range(43)
                    for ptc in string.printable:
                        x1 = i % p
                        x2 = (ord(ptc)**q) % (q**2 - 6 * q + 6)
                        x3 = key[7 * i % len(key)]
                        enc = x1 ^ x2 ^ x3
                        if enc == ord(data[i]):
                            pt += ptc
                print pt
```

The flag is:

```
SharifCTF{6494889069126bb688b8755815a8d672}
```

© 2017 GitHub, Inc. Terms Privacy Security Status Help

Contact GitHub API Training Shop Blog About