 **grocid / CTF**

Watch9

Star29

Fork6

<> Code

🔔 Issues 1

🔗 Pull requests 0

📁 Projects 0


📡 Pulse

📊 Graphs

Branch: master ▾





CTF / Hack the vote / 2016 /

Create new fileFind fileHistory

 grocid kek top

Latest commit d2a45ee on 7 Nov 2016

..

 README.md	Hack the vote	5 months ago
 decrypted.gif	kek top	5 months ago
 trump.jpg	Hack the vote	5 months ago
 trump1.jpg	Hack the vote	5 months ago

 README.md

TOPKEK (50 p)

A CNN reporter had only one question that she couldn't get off her mind

Do we even know, who is this 4 CHAN???

So she set out to find who this 400lb hacker is. During her investigation, she came across this cryptic message on some politically incorrect forum online, can you figure out what it means?

There is a ciphertext attached to this challenge, which contains the following:

```
KEK! TOP!! KEK!! TOP!! KEK!! TOP!! KEK! TOP!! KEK!!! TOP!! KEK!!!! TOP! KEK! TOP!! KEK!! TOP!!! KEK! TOP!!!! KEK! TOP
```

First guess is a binary, and judging from the amount of points given for this challenge, that is probably no more complicated than that. TOP is either 0 or 1 and the number of ! denotes the number of each symbol. We try both and find that TOP maps to 1, so KEK is 0. The following code decodes the ciphertext.

```
split_cipher = ciphertext.split()
decrypted = ''
for block in split_cipher:
    if block.startswith('KEK'):
        decrypted += '0' * (len(block) - 3)
    else:
        decrypted += '1' * (len(block) - 3)

decrypted = int(decrypted, 2)
print libnum.n2s(decrypted)
```

This gives

```
flag{T0o0o0o0o0P_____1m_h4V1nG_FuN_r1gHt_n0W_4R3_y0u_h4v1ng_fun_____K3K!!!}
```

Trump Trump (100 p)

With Trump about to be in office, autographed photos of him are selling like wildfire. The only problem is: Trump makes it a point to never sign a photo of himself. If you could get a signed picture, you could stand to make DOZENS of dollars.

```
nc trumptrump.pwn.republican 3609
```

We get the following parameters

```
e = 65537
N = 23377710160585068929761618506991996226542827370307182169629858568023543788780175313008507293451307895240053109844
```

Converting the image to a number

```
f = open('trump.jpg', 'r')
data = libnum.s2n(f.read()) % N
```

we try to submit it, which incidentally fails. We can exploit that $\text{sign}(a) \times \text{sign}(b) = \text{sign}(a \times b)$. Since the image is divisible by for instance 5, we can factor out this and sign the two parts separately.

```
sign(data / 5) = 1574210524795873695800485984486010639265052964249144479165528865305913980005320616702379287615933833
sign(5) = 18938431620064949405099081881389422411569506620645684785718437650149907701313939238017399264771270907473551
```

so,

```
sign(data) = 72405272600441268990758323399732559239433543350600370015581053432954958416358436035074119964885689772061
```

Sending this to the service, we get

```
Duneld Trump: Well, I already met my quota for Guantanamo Bay inmates today, so okay.
Trump looks shocked, appalled by the fact that he'd sign a picture of himself for such a not-billionaire.
Trump sprints away at a blinding 2mph, dropping what he was carrying.
It's a stack of photos, you pick one up and look at it.
ffd8ffe000104a46494600010101006000600000ffdb004300030202030202030303040304050805050404050a070706080c0a0c0c0b0a0b0c
```

Let us write code to make it viewable:

```
import libnum
from pwn import *

v = remote('trumptrump.pwn.republican', 3609)
v.recv(1024)
v.send(str(s) + '\r\n')

for i in range(4): v.recvline()
data = v.recvline()
open('trump1.jpg', 'w').write(libnum.n2s(int(data, 16)))
```



Vermatrix Supreme (100 p)

Working in IT for a campaign is rough; especially when your candidate uses his password as the IV for your campaign's proprietary encryption scheme, then subsequently forgets it. See if you can get it back for him. The only hard part is, he changes it whenever he feels like it.

```
nc vermatrix.pwn.democrat 4201
```

Along this, we get the code for the encryption scheme.

```
import sys, random, time

flag = "flag{1_sw34r_1F_p30P13_4cTu41Ly_TrY_Th1s}"

def printmat(matrix):
    for row in matrix:
        for value in row:
            print value,
        print ""
    print ""

def pad(s):
    if len(s)%9 == 0:
        return s
    for i in xrange((9-(len(s)%9))):
        s.append(0)
    return s

def genBlockMatrix(s):
    outm = [[7 for x in xrange(3)] for x in xrange(3)] for x in xrange(len(s)/9)]
    for matnum in xrange(0, len(s)/9):
        for y in xrange(0, 3):
            for x in xrange(0, 3):
                outm[matnum][y][x] = s[(matnum*9)+x+(y*3)]
    return outm

def fixmatrix(matrixa, matrixb):
    out = [[0 for x in xrange(3)] for x in xrange(3)]
    for rn in xrange(3):
        for cn in xrange(3):
            out[cn][rn] = (int(matrixa[rn][cn])|int(matrixb[cn][rn]))&~(int(matrixa[rn][cn])&int(matrixb[

    return out

def chall():
    IV = [c for c in '?????????']
    seed = "?????????????????"

    blocks = genBlockMatrix(pad(IV + [ord(c) for c in seed]))

    res = [[0 for i in xrange(3)] for i in xrange(3)]
    for i in xrange(len(blocks)):
        res = fixmatrix(res, blocks[i])

    print "SEED: " + str(seed)
    printmat(res)

    data = raw_input("")

    data = data.replace(' ', '').strip().split(',')

    if len(data) != 9:
        return False

    for i in xrange(len(IV)):
        if str(IV[i]) != str(data[i]):
            return False

    return True

if chall():
    print flag
```

The `iv` is a series of 9 numbers in some undefined range. Instead of spending a bunch of time to find some algebraic relations that we can exploit, we can use Z3. We define a set of variables (BitVec) for Z3, representing the unknown `iv`. Since the matrix we receive should contain the same values as the computed one, given `seed` and the correct `iv`, we set up equality relations.

```
from z3 import *
from pwn import *
from vermatrix import *

def find_iv(seed, target):
    s = Solver()
    IV = [BitVec('%d' % i, 32) for i in range(1,10)]

    blocks = genBlockMatrix(pad(IV + [ord(c) for c in seed]))
    res = [[0 for i in xrange(3)] for i in xrange(3)]

    for i in xrange(len(blocks)):
        res = fixmatrix(res, blocks[i])

    for y1, y2 in zip(res, target):
        for x1, x2 in zip(y1, y2):
            s.add(x1 == x2) # they should be equal!

    return s
```

Now, we can use this routine in connecting to the server as follows:

```
context.log_level = 'error'
v = remote('vermatrix.pwn.democrat', 4201)

seed = v.recvline().split()[1]
target_matrix = [[int(x) for x in v.recvline().split()] for i in range(0, 3)]
s = find_iv(seed, target_matrix)

if s.check() == sat:
    m = s.model()
    out = [0]*9
    for x in m:
        out[int(str(x))-1] = m[x]
    v.send(str(out)[1:-1] + '\n')
    print 'FLAG: {}'.format(v.recvline())
```

Running the whole code, we obtain

```
flag{IV_wh4t_y0u_DiD_Th3r3}
```

Boxes of Ballots (200 p)

Privjet Komrade!

While doing observing of Amerikanski's voting infrascture we find interesting box. We send operative to investigate. He return with partial input like showing below. He say box very buggy but return encrypted data sometimes. Figure out what box is do; maybe we finding embarass material to include in next week bitcoin auction, yes?

```
ebug": true, "data": "BBBBBBBBBBBBBBBB", "op": "enc"}
```

```
nc boxesofballots.pwn.republican 9001
```

This challenge is very similar to [this](#). We can solve it as follows:

```
from pwn import *
import json
import string
```

```

def getdata(res):
    for l in res.split('\n'):
        if l.startswith('{"Status"}'):
            return json.loads(l)['data']

def getreference(buflen, v):
    data = {"debug": False, "data": 'x'*buflen, "op": "enc"}
    payload = json.dumps(data).replace('False', 'false')
    v.send(payload + '\n')
    response = v.recv(1024)
    ref = getdata(response)[:64]
    return ref

context.log_level = 'error'
v = remote('boxesofballots.pwn.republican', 9001)
result = ""
buflen = 31

while buflen:
    ref = getreference(buflen, v)

    for i in string.printable:
        data = {"debug": False, "data": 'x'*buflen + result + i, "op": "enc"}
        payload = json.dumps(data).replace('False', 'false')

        v.send(payload + '\n')
        response = v.recv(1024)
        res = getdata(response)
        if res[:64] == ref:
            result += i
            buflen -= 1
            print result
            break
    else:
        print "[-] Error"
        exit(-1)

```

gives

```
flag{Source_iz_4_noobs}
```

Michał Żuberek ([Z](#)) of my team Snatch the Root solved this one.

The Best RSA (250 p)

At his last rally, Trump made an interesting statement:

I know RSA, I have the best RSA
 The more bits I have, the more secure my cyber, and my modulus is YUUUUUUUUUUUUUGE
 We don't believe his cyber is as secure as he says it is. See if you can break it for us

We get a file with a public exponent $e = 65537$ and a massive public modulus n . Clearly, there is something strange with this modulus. The first obvious move is to check for small factors, and in fact, we find that 3 is a factor. Not once, but several times. The whole modulus consist of very small prime factors. Let us write some code to factor it!

```

import libnum, grocid, challenge

def get_private_exponent(phi, e):
    return libnum.modular.invm(e, phi)

def find_factors(n, h):
    primes = grocid.sieve(h)
    factors = {}
    for p in primes:
        while n % p == 0:
            n = n / p
            if p in factors:
                factors[p] += 1
            else:
                factors[p] = 1

```

```

    return factors

print '[+] Factoring n...'
single_factors = find_factors(challenge.n, 10000)

```

OK, so we got a dictionary containing each prime factor and its corresponding exponent. It looks like this:

```
{3: 1545, 5: 1650, 7: 1581, 137: 1547, 11: 1588, 13: 1595, 17: 1596, 19: 1553, 149: 1572, 23: 1579, 29: 1549, 31: 161
```

So, we can compute ϕ and the corresponding private exponent:

```

phi = challenge.n
print '[+] Computing phi...'
for p in single_factors:
    phi = phi / p * (p - 1)

print '[+] Computing private exponent'
d = get_private_exponent(phi, 65537)

```

This enables us to decrypt! However, if you try it, you will probably notice that it takes quite some time and memory to compute $c^d \pmod n$. Now, we can speed this a bit using CRT. So, we compute $c^d \pmod{3^{1545}}$ and so on separately, for each prime factor and then use CRT to reconstruct the whole message:

```

remainders = []
moduli = []

for p in single_factors:
    modulus = pow(p, single_factors[p])
    moduli.append(modulus)
    remainder = pow(challenge.c, d, modulus)
    remainders.append(remainder)

plaintext = libnum.modular.solve_crt(remainders, moduli)
print 'Message: {0}'.format(libnum.n2s(plaintext))

```

It still takes some time to complete, but it is manageable (left it running while doing other stuff so...). It could possibly be made faster by Hensel lifting, but I decided not to try it. When done, we get a message which is a image file.:



Baby's hands (300 p)

We think that Trump's right hand man has been sending out flags from his personal computer, but we need to be sure. See if you can make anything out of the traffic we intercepted.

The initial two lines are

```

{d:n:c}
{64193765095472280945778947695026260940793161700792092928929371930940586875921621250436677664062645637750266086941626

```

OK, so the private exponent d is published. We try to compute $c^d \pmod n$ and look at the binary data with binwalk. No results. Hm... so, what if we need the public exponent e to compute the plaintext? We can probably find this with Wiener's attack! Let us try it!

```
import libnum, grocid

f = open('intercepted', 'r')

data = f.read()
data = data.split('\n')

for line in data[1:]:
    d, n, c = [int(x) for x in line[1:-1].split(':')]
    e = grocid.wiener.attack(d, n)
```

This yields

```
flag{G3t_1t?_1t_h4s_4_sm4ll_d}
```

SMTPresident (400 p)

The FBI reopened their investigation of Hillary Clinton after they recovered some interesting files from her personal email server.

```
emails
pubkeys
flag
```

The file `pubkeys` contains a set of public keys with very small public exponent ($e = 17$). Assuming that every message sent on the same date is identical, we can use Håstad's broadcast attack. Incidentally, we have 17 of each every date. The following Python code will do just fine:

```
from os import listdir
from os.path import isfile, join
from Crypto.PublicKey import RSA
from base64 import b64decode
import re, libnum, gmpy

def get_files(path):
    return [f for f in listdir(path) if isfile(join(path, f))]

pubkeysfiles = get_files('./pubkeys')
emailfiles = get_files('./emails')
pubkeys = {}
encrypted_emails = {}

for pubkeyfile in pubkeysfiles:
    f = open('./pubkeys/' + pubkeyfile, 'r')
    data = f.read().replace('-----BEGIN RSA PUBLIC KEY-----', '').replace('-----END RSA PUBLIC KEY-----', '')
    data = b64decode(data)
    pubkeys[pubkeyfile] = RSA.importKey(data)
    f.close()

for emailfile in emailfiles:
    f = open('./emails/' + emailfile, 'r')
    data = f.read()
    sender = re.findall('To: (.*?)@dnc', data)[0]
    date = re.findall('Date: (.*?)\n', data)[0]
    content = re.findall('Content: (.*?)', data)[0]
    if not date in encrypted_emails:
        encrypted_emails[date] = {}
    encrypted_emails[date][sender] = int(content, 16)

message_content = False
for date in encrypted_emails:
    moduli = []
    remainders = []
```

```
for x in encrypted_emails[date]:
    moduli.append(pubkeys[x].n)
    remainders.append(encrypted_emails[date][x])

x = libnum.modular.solve_crt(remainders, moduli)
x0 = gmpy.mpz(x)
xr = x0.root(17)
message = libnum.n2s(int(xr[0]))

if not message_content:
    message_content = ['?'] * len(message)

for i, c in enumerate(message):
    if c != '#':
        message_content[i] = c

print ''.join(message_content)
```

which outputs

```
Subject: My Fellow DNC Members
Content: Keep this safe <MISSING>186286610343108349347724171711756660997909706467024801180047812829348705350016982496
```



OK, so let us assume that the key mentioned in the message is the lower bits of the private exponent. There is a method to solve this problem. A script which implements this method can be found [here](#) and [here](#).

Unfortunately, the computation did not finish (e = 65537 takes some time to run, even multithreaded). Pity.

