笑脸漏洞渗透linux,TWCTF 2018 escapeme writeup

转载

范汝诗 于 2021-05-16 13:01:14 发布 240 收藏
文章标签： 笑脸漏洞渗透linux

这是我之前见过最好的KVM题了。感谢@shift_crops出的这么吊的题，赛后还放了源码。

原题文件可以在作者的repo里找到，包括4个二进制文件，两个文本文件和一个python脚本。

我的CTF writeup repo里有我三个exp。

介绍

题目有三个文件，kvm.elf, kernel.bin和memo-static.elf。

这个题里有三个flag，分别需要用户空间任意shellcode执行，内核空间执行和host模拟器(kvm.elf)执行。

shell里输入./kvm.elf kernel.bin memo-static.elf跑起来，会看到一个看起来比较常规的pwn题的样子：

kvm.elf 是一个模拟器(和qemu-system一个作用)，装备了kvm，也就是一个在linux内核里实现的vm，用来做模拟。

kernel.bin实现了一个非常小的内核，可以加载静态的ELF文件，以及有一些系统调用。

memo-static.elf是一个实现备忘录系统的常规ELF文件。

因为作者的repo里已经把源码放出来了，我这就不再介绍整个题了，就主要关注漏洞部分。

EscapeMe1: 用户空间

memo-static.elf是一个静态链接的文件，checksec结果：

其实对这个题来说checksec 基本没什么卵用，因为用来执行这个二进制文件的“内核”是由kernel.bin来实现的，其实根本就没开针对可执行文件的任何现代保护方法，导致的结果就是没有aslr, 没有NX(所有段都可执行),只要我们能控制rip就万事大吉了。

漏洞

bug非常容易看到，在Alloc里我们可以在堆上加一个memo，最多可以有0x28个字节的数据，之后我们可以有一次Edit各个memo的机会，edit的实现如下：

read(0, memo[id].data, strlen(memo[id].data));

如果一个memo刚好有0x28个不带null字节的数据，那么这个地方就可以溢出到下一个chunk了。

利用

这题并不难，不过有个问题就是这个内存分配器不是我们在glibc里熟悉的那个ptmalloc。这里的malloc/free的实现其实和ptmalloc非常像，不过没有tcache和fastbin。

我们采用的方法是用伪造chunk进行unlink，如下图：

|----------|

| | 0x31 |

(*ptr) -> | | 0x51 |

| ptr - 0x18 | ptr - 0x10 |

|----------|

| | 0x31 |

| BBBBBBBB | BBBBBBBB |

| BBBBBBBB | BBBBBBBB |

|----------|

伪造 prev size -> | 0x50 | 0x30 |

| CCCCCCCC | CCCCCCCC |

| CCCCCCCC | CCCCCCCC |

|----------|

堆溢出发生在B块里，把下一块的size从0x31改为0x30，然后也构造一个prev_size(0x50)。

之后我们Delete(free)掉C块，这样就会试图把前块(伪造的)unlink掉。最后，本来指向堆的*ptr就可以指向ptr - 0x18。

之后，我们几乎就可以任意写了，但是因为我们必须写同样长度的数据，所以还是挺限制的。(回忆一下之前的Edit实现)。所以现在我们还不能直接修改栈上的数据(因为地址0x7fffffffxx比heap地址0x606000要长).我在这其实卡了一会，最后用的方法：

把top_chunk(位于0x604098)指针修改为0x604038。

改为0x604038是因为这有个0x604040的值，所以在malloc的时候可以过size的check

Alloc三个memo，第三个就会分配在top_chunk自己这，然后我们伪造top_chunk，使其指向一个栈地址

再Alloc一个memo，然后我们就可以分配在栈上了，之后就可以伪造返回地址了。

之后就是把rip改成准备好的shellcode，用来读下一步的shellcode，然后执行。

之后，我就又卡住了。[笑哭]

是，我现在是有任意代码执行了，但是flag呢？由于我觉得要继续pwn其他的部分(内核和模拟器部分)，我们反正都得先做到代码执行，所以我就直接去尝试继续去利用，压根没去拿flag1.

之后我稍微逆了一下，发现在kernel.bin的实现里有一个0x10c8为调用号的特殊系统调用。这个系统调用把flag拷到了一个只写的页里：

uint64_t sys_getflag(void){

uint64_t addr;

char flag[] = "Here is first flag : "FLAG1;

addr = mmap_user(0, 0x1000, PROT_WRITE);

copy_to_user(addr, flag, sizeof(flag));

mprotect_user(addr, 0x1000, PROT_NONE);

return addr;

}

于是我们只需要调用一下这个系统调用，然后mprotect一下让这个页可以读，然后打印出来就可以了

shellcode = asm('''

mov rax, 0x10c8

syscall

mov rbp, rax

''' + shellcraft.mprotect('rbp', 0x1000, 6) + shellcraft.write(1, 'rbp', 60))

比赛期间我写的exploit可以在我的github repo里找到。

其实我当时都没注意到NX没开，所以我是ROP，mmap了一个新page来放shellcode的。所以其实这个利用比我这讲的要麻烦一点。

Flag1:

TWCTF{fr33ly_3x3cu73_4ny_5y573m_c4ll}

EscapeMe2: 内核空间

kernel.bin包括三个部分：

实现了一个解析和加载用户程序的简单execve

实现了一个MMU表，用来把虚拟内存转换到物理内存。

实现了几个系统调用，包括：read, write, mmap, munmap, mprotect, brk, exit 和 get_flag(给EscapeMe1用的)

我和队友花了点时间来找内存相关操作，比如mmap, munmap和MMU的实现部分的漏洞，发现根本就不对。

我们的目标当然是做到内核层的shellcode任意执行。但是因为这个自己写的MMU表把虚拟地址是否能够由用户空间访问用一个bit标记了一下，所以我们不能直接用用户空间shellcode去重写kernel代码。

漏洞

根据hint，我们知道在内存管理部分是有个洞的。

bug是由在模拟器和内核间的abi不一致造成的。在模拟器里有一个自己实现的内存分配器，palloc和pfree，然后kernel把pfree用挫了。

在用户调mmap(vaddr, len, perm)系统调用的时候，内核会：

hyper call调用palloc(0, len)，来获取一个物理地址paddr，长度为len。

设置好MMU表，把vaddr映射到paddr，并且把权限位设置好。设置期间可能会调用一些palloc(0, 0x1000)(这得看vaddr相应的entry是否已经创建了)

返回vaddr

而在用户调用munmap(vaddr, len)的时候，内核会：

把vaddr映射到paddr

hyper call调用到for(i = 0 ~ len >> 12) pfree(paddr + (i << 12), 0x1000)

这里其实只要pfree像内核想的这样工作的话就没问题的。

在模拟器里，pfree(addr, len)压根就不关心len(他的函数圆形是pfree(void*)))

所以，如果有长度为0x2000的内存addr，然后调用munmap(addr, 0x1000)，内核其实只把第一页unmap了，但是模拟器里，整个内存都被free了！

再说明白点的话，之前的代码大概这样：

shellcode = asm(

mmap(0x7fff1ffc000, 0x2000) +

munmap(0x7fff1ffc000, 0x1000) +

mmap(0x217000, 0x1000)

)

在这段shellcode被执行之后，0x7fff1ffc000 + 0x1000还是可以被用户访问，但是已经指向了刚才映射0x217000的时候palloc的MMU entry了！

利用

如果我们能够伪造MMU表的话，那整个事情就简单了。在一些设置之后，我的0x217000映射到了物理内存0x0，也就是内核代码的地址。

现在我们只需要调用个read(0, 0x217000+off, len)就可以改掉内核部分了。

在模拟器里有个比较有用的hyper call调用可以把一个文件读到buffer里，用这个就可以很简单的拿到flag2.txt了。

kernel_sc = asm('''

mov rdi, 0

call sys_load_file

movabs rdi, 0x8040000000

add rdi, rax

mov rsi, 100

call sys_write

ret

sys_write:

```
mov eax, 0x11

mov rbx, rdi

mov rcx, rsi

mov rdx, 0

vmmcall

ret

sys_load_file:

mov eax, 0x30

mov ebx, 2 /* index 2, the flag2.txt */

mov rcx, rdi /* addr */

mov esi, 100 /* len */

movabs rdx, 0x0

vmmcall

ret

''')
```

这一部分的完整脚本在这

Flag2:

TWCTF{ABI_1nc0n51573ncy_l34d5_70_5y573m_d357ruc710n}

EscapeMe3: 操纵世界

现在就剩最后一步了：把模拟器pwn掉。

首先我们得先看看开了那些seccomp的规则，如果想去pwn模拟器的话。

漏洞

在EscapeMe2里我们已经可以伪造MMU表了，这个阶段也会用到这个。MMU表里的物理地址record其实是在模拟器里mmap的页的offset，也就是刚好在libc-2.27.so前面的页。所以说我们有MMU表的伪造能力，就可以访问到glibc里的内存。

而且我在题放出来5分钟之内就发现了seccomp规则里有个bug，这里用到了我吊的不行的工具seccomp-tools [大笑脸]。

Seccomp-tools的模拟器清晰的告诉我们满足args[0] & 0xff < 7的系统调用都能用。

之后就没啥新东西了，直接pwn掉就行了。

利用

通过伪造MMU表我们可以做到任意内存访问，但是需要先干掉ASLR。通过读libc里的指针可以同时leak出libc的基地址和argv地址，之后就可以往栈上写ROP链了。

ROP链主要用来调用mprotect(stack, 0x3000, 7)然后跳到栈上的shellcode。

因为有seccomp的限制，所以在execve之后的syscall，比如说open都没法用，我们就没法起shell，所以我选择写了个ls的shellcode来获取flag3的文件名。

```
asm('''
/* open('.') */
mov rdi, 0x605000
mov rax, 0x2e /* . */
mov [rdi], rax
mov rax, 2
xor rsi, rsi
cdq
syscall
/* getdents */
mov rdi, rax
mov rax, 0x4e
mov rsi, 0x605000
cdq
mov dh, 0x10
syscall
/* write */
mov rdi, 1
mov rsi, 0x605000
mov rdx, rax
mov rax, 1
syscall
'''))
```

输出：

之后再读文件flag3-415254a0b8be92e0a976f329ad3331aa6bbea816.txt就可以拿到最终flag了。

Flag3:

TWCTF{Or1g1n4l_Hyp3rc4ll_15_4_h07b3d_0f_bug5}

结论

从这个题里我学了不少KVM的知识(虽然好像对这题来说没啥卵用)，然后这种一层一层逃逸的设计还是很不错的，挺好玩。

我之后还会写篇文章讲讲KVM是怎么工作的，一方面帮我自己记一下，二方面也可以作为KVM新手介绍。

再次感谢@shift_crops让我这周末玩的挺开心 □