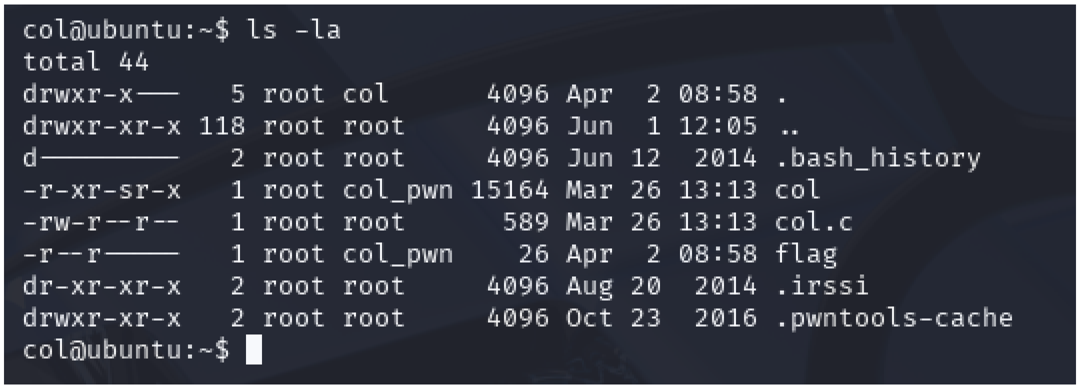
# collision - pwnable.kr

ssh col@pwnable.kr -p2222 (pw:guest)

This looks like it will focus on hash collisions. Lets take a look at what we have in the machine



```
col@ubuntu:~$ ls -la
total 44
drwxr-x---   5 root col      4096 Apr  2 08:58 .
drwxr-xr-x 118 root root     4096 Jun  1 12:05 ..
d---------   2 root root     4096 Jun 12  2014 .bash_history
-r-xr-sr-x   1 root col_pwn 15164 Mar 26 13:13 col
-rw-r--r--   1 root root      589 Mar 26 13:13 col.c
-r--r-----   1 root col_pwn    26 Apr  2 08:58 flag
dr-xr-xr-x   2 root root     4096 Aug 20  2014 .irssi
drwxr-xr-x   2 root root     4096 Oct 23  2016 .pwntools-cache
col@ubuntu:~$ 
```

I want to take a look at what's inside col.c

Running the cat command, I get to see the following code

```
#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] )){
                setregid(getegid(), getegid());
                system("/bin/cat flag");
                return 0;
        }
        else
                printf("wrong passcode.\n");
        return 0;
}
```

So the code checks for a 20 byte password and then compares it with the hashcode by running the check_password function. In that function, p gets cast as an integer which is 4 bytes. This converts the 20 byte password string to the variable ip which is an array of 5 indices of 4 bytes. Adding all the index values together should result in the same value as the hashcode.

## Approach

We have to be weary to not pass in null bytes when passing in hex values. There are many different combinations that sum up to 0x21DD09EC, and the first approach was to subtract 0x4444444 so that I would have 4 chunks of 0x1111111 and the rest in the last chunk

21DD09EC – 1111111 = 20CBF8DB

20CBF8DB – 1111111 = 1FBAE7CA

1FBAE7CA – 1111111 = 1EA9D6B9

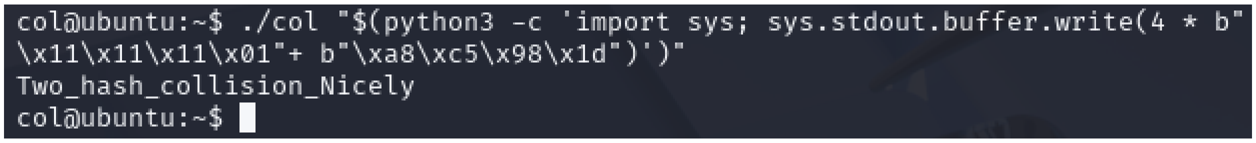
1EA9D6B9 – 1111111 = 1D98C5A8

Now to check the math 0x1D98C5A8 + (4 ∗ 0x01111111) = 0x21DD09EC seems correct.

I injected the hex values using python and formatted them to little endian

```
./col "$(python3 -c 'import sys; sys.stdout.buffer.write(4 * b"\x11\x11\x11\x01"+ b"\xa8\xc5\x98\x1d")')"
```

Once running the code, we received our flag



```
col@ubuntu:~$ ./col "$(python3 -c 'import sys; sys.stdout.buffer.write(4 * b"
\x11\x11\x11\x01"+ b"\xa8\xc5\x98\x1d")')"
Two_hash_collision_Nicely
col@ubuntu:~$ 
```

## Additional Thoughts

The reason that this challenge is called collision is due to the fact that there are many different unique combinations that result in the same hash. When we do hashing, we want all unique items we hash to have a unique hash value for integrity.

To show that, we can approach the question with division instead of subtraction:

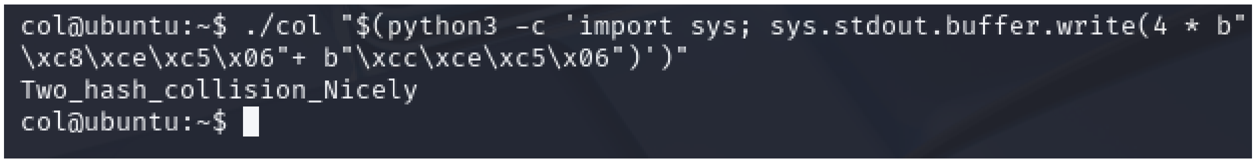
21DD09EC / 5 = 6C5CEC8 Remainder  4

Therefore we can have 4 chunks of 6C5CEC8 and one with the added remainder to it, which is 6C5CECC

The code would then look like:

```
./col "$(python3 -c 'import sys; sys.stdout.buffer.write(4 * b"\xc8\xce\xc5\x06"+ b"\xcc\xce\xc5\x06")')"
```

And we can see that this was also successful



```
col@ubuntu:~$ ./col "$(python3 -c 'import sys; sys.stdout.buffer.write(4 * b"
\xc8\xce\xc5\x06"+ b"\xcc\xce\xc5\x06")')"
Two_hash_collision_Nicely
col@ubuntu:~$ 
```