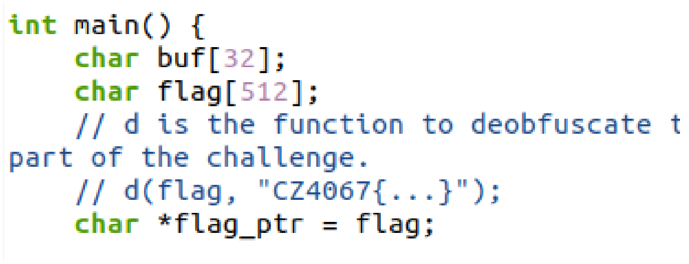
From the C code we find that at line 18, printf(buf) is susceptible to format string attack.

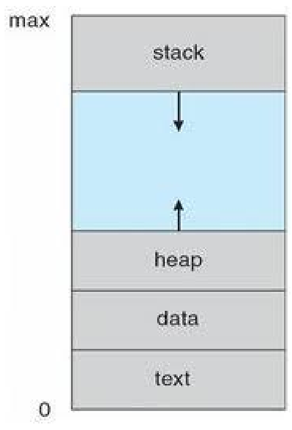
In the program, after the flag is initialized, a character pointer *flag_ptr is created to point to the flag string.



```
int main() {
    char buf[32];
    char flag[512];
    // d is the function to deobfuscate t
part of the challenge.
    // d(flag, "CZ4067{...}");
    char *flag_ptr = flag;
```

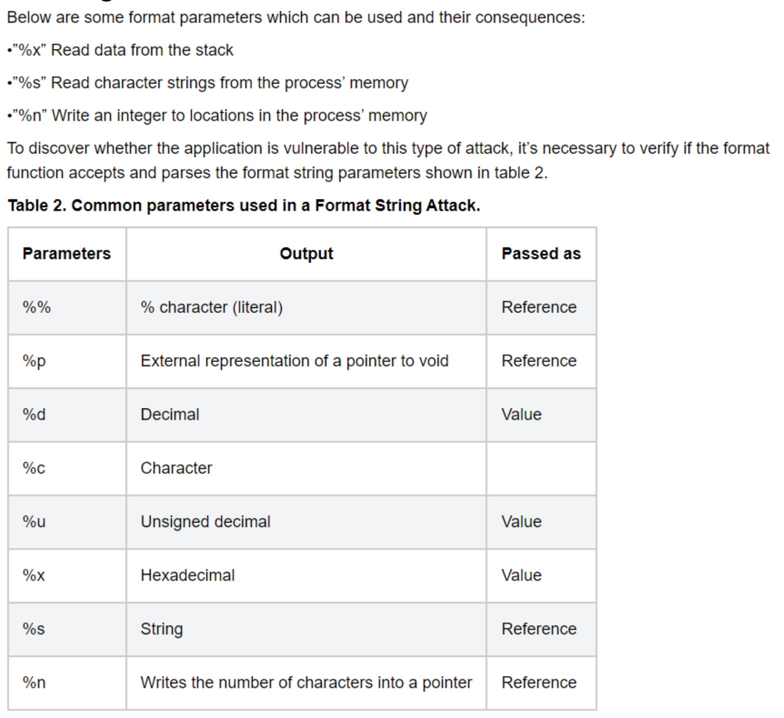
The **buf** array and the **flag** array are both allocated on the stack, and **flag_ptr** is a local variable that holds a pointer to the flag array.

The order in which these variables are allocated on the stack will depend on the specific implementation of the compiler and the size of the arrays. However, in general, local variables are allocated on the stack in the *reverse order* in which they are declared.



In general, the stack grows downward, meaning that the most recently allocated variables will be located at the top of the stack. Therefore, if flag_ptr is the last local variable to be allocated in main(), it is likely to be located at or near the top of the stack.

According to OWASP:



Below are some format parameters which can be used and their consequences:

•"%x" Read data from the stack

•"%s" Read character strings from the process' memory

•"%n" Write an integer to locations in the process' memory

To discover whether the application is vulnerable to this type of attack, it's necessary to verify if the format function accepts and parses the format string parameters shown in table 2.

Table 2. Common parameters used in a Format String Attack.

Parameters	Output	Passed as
%%	% character (literal)	Reference
%p	External representation of a pointer to void	Reference
%d	Decimal	Value
%c	Character	
%u	Unsigned decimal	Value
%x	Hexadecimal	Value
%s	String	Reference
%n	Writes the number of characters into a pointer	Reference

%s will not give us the flag, instead it will lead to segmentation fault. Reason: When we call printf(buf), the buf array is used as the format string for printf(). This means that printf() will try to interpret the contents of buf as a format string, and print out values from the stack according to the corresponding format specifiers.

Inputting %s as the format specifier, printf() will try to interpret the value of buf as a pointer to a null-terminated string, and print out the string it points to. Since the value of buf is not a valid memory address and does not point to a null-terminated string, the behavior is undefined and can result in unpredictable output or a segmentation fault.

Therefore, we should specify which parameter on the stack we would like to read from.

According to MIT, in X86-64 Architecture, parameters to a function are typically stored in registers or on the stack.

On 64-bit architectures such as x86-64, there are several general-purpose registers that can be used for passing function arguments, including:

- RDI (or EDI): used for the first parameter
- RSI (or ESI): used for the second parameter
- RDX (or EDX): used for the third parameter
- RCX (or ECX): used for the fourth parameter
- R8: used for the fifth parameter
- R9: used for the sixth parameter
- R10: General-purpose register
- R11: General-purpose register
- R12-R15: General-purpose registers
- XMM0-XMM15: Registers for floating-point operations or SIMD (Single Instruction, Multiple Data) instructions

After the 6th parameter, we can call %7 -> top of stack -> *flag pointer to get flag.

Therefore input %7$s, and the flag will show.