Problem 1 It is common in C programs to see memory allocation idioms such as

```c
struct s *sarray = malloc( nelm * sizeof(struct s) );
if (sarray == NULL) exit(EXIT_FAILURE);
/* else use sarray[0], sarray[1], etc. */
```

where the (unsigned) number of elements `nelm` is derived from program input.

(a) What is the problem with this idiom? Under what circumstances can the attacker cause the program to misbehave, and what can he achieve as a result? Be specific.

(b) Write a function `alloc_array` that is safe to use instead. The function should call `malloc` when it is safe to do so, and return `NULL` when it is not. The function prototype should be `void *alloc_array(size_t nelm, size_t elmsize)`.

You may assume a 32-bit system, with the following definitions:

```c
typedef uint32_t size_t;
#define SIZE_MAX 0xffffffffUL
void *malloc(size_t size);
```

Your solution shouldn’t assume compiler support for 64-bit `long long` integer types.
Problem 2 In this problem, we will consider vulnerability exploitation in the presence of perfect control-flow integrity. Any attempt by the attacker to modify a function pointer or saved return address will be prevented by this (hypothetical) perfect CFI.

Consider the following flagrantly vulnerable program fragment:

```c
FILE *logfile = fopen(...);
char formatbuf[80] = "Processed a line.\n";
char buf[80];

while (NULL != gets(buf))
    fprintf(logfile, formatbuf);
```

The unsafe `gets` call allows the attacker to overwrite both `buf` and `formatbuf` with any bytes she wants, including the NUL byte (but excluding '\n'). We will assume that the vulnerable function’s stack frame is laid out so that `fprintf` expects the first format parameter at the very beginning of `buf`. The adversary thus controls both the format string and the format arguments, and may cause the loop to repeat as many times as she wishes.

This is enough to allow the adversary to induce the program to perform arbitrary computation! The adversary will encode boolean values as follows. A true will be encoded as the little-endian value 0x01 0x00 0x00 0x00 (that is, a 1 byte followed by three 0 bytes). A false will be encoded as the little-endian value 0x00 0x00 0x00 0x00 (that is, four 0 bytes).

(It may be useful to know that if `printf` encounters an ASCII character with value 1, 2, 254, or 255, it will just echo it to the output, the same as any printable character like '\x'.)

(a) Let `in1` and `in2` be the addresses of two memory locations that hold encoded boolean values, and let `out` be the address of another memory location. Consider the format string

```
"%s%s%n"
```

supplied with the arguments (1) the address `in1`; (2) the address `in2`; (3) the address `out`; and followed by the format string

```
"%s%n"
```

supplied with the arguments (1) the address `out`; (2) the address `out`.

Explain why, if these two format strings and corresponding format arguments are formatted by `printf`, they will cause the memory at address `out` to contain the boolean or of the values at addresses `in1` and `in2`, encoded using the encoding given above.
(b) The approach above used two format strings, but that isn’t absolutely necessary. Let \( in \) be the address of a location in the program’s memory that holds a boolean value encoded as above, and let \( out \) be the address of another location in the program’s memory. Consider the format string

\[
\text{"%a%255d%s%hhn%s%s%hhn"}
\]

supplied with the following arguments: (1) the address \( out \); (2) the constant value 0; (3) the address \( in \); (4) the address \( out \); (5) the address \( out \); (6) the address \( out \); (7) the address \( out \).

Explain why, if this format string and corresponding format arguments are formatted by printf, they will cause the memory at address \( out \) to contain the boolean negation of the memory at address \( in \), encoded using the encoding given above. (Hint: there are just two cases; work through both.)

Since not and or together are logic-complete, we can encode any logical function as a series of printf calls; a similar printf trick allows us to perform conditional assignments.

(c) Suppose we wish to exploit a program that allows us one printf where we control the format string and arguments, rather than arbitrarily many. What memory locations should our format string overwrite to allow itself to loop back and execute again from the beginning? Be specific.

(Receive that we still assume perfect CFI, so we cannot overwrite any function points or saved return addresses.)
NAME
fprintf -- formatted output conversion

LIBRARY
Standard C Library (libc, -lc)

SYNOPSIS
#include <stdio.h>

int
fprintf(FILE * restrict stream, const char * restrict format, ...);

DESCRIPTION
The printf() family of functions produces output according to a format as
specified below. The fprintf() function writes output to the
given output stream.

These functions write the output under the control of a format string
that specifies how subsequent arguments are converted for output.

The format string is composed of zero or more directives: ordinary char-
acters (not %), which are copied unchanged to the output stream; and con-
version specifications, each of which results in fetching zero or more
subsequent arguments. Each conversion specification is introduced by the
% character. The arguments must correspond properly (after type promo-
tion) with the conversion specifier. After the %, the following appear in
sequence:

o An optional decimal digit string specifying a minimum field width.
  If the converted value has fewer characters than the field width, it
  will be padded with spaces on the left (or right, if the left-adjust-
  ment flag has been given) to fill out the field width.

o An optional length modifier, that specifies the size of the argument.
  The following length modifiers are valid for the d, i, n, o, u, x, or
  X conversion:

Modifier    d, i  o, u, x, X  n
hh signed char  unsigned char  signed char *
h short      unsigned short   short *

The conversion specifiers and their meanings are:

diouux The int (or appropriate variant) argument is converted to signed
decimal (d and i), unsigned octal (o), unsigned decimal (u), or
unsigned hexadecimal (x and X) notation. The letters ‘‘abcdef’’
are used for x conversions; the letters ‘‘ABCDEF’’ are used for X
conversions. The precision, if any, gives the minimum number of
digits that must appear; if the converted value requires fewer
digits, it is padded on the left with zeros.

s The char * argument is expected to be a pointer to an array of
character type (pointer to a string). Characters from the array
are written up to (but not including) a terminating NUL charac-
ter; if a precision is specified, no more than the number speci-
fied are written. If a precision is given, no null character
need be present; if the precision is not specified, or is greater
than the size of the array, the array must contain a terminating
NUL character.

n The number of characters written so far is stored into the inte-
ger indicated by the int * (or variant) pointer argument. No
argument is converted.