Question 1 ***Memory safety***

Q1.1 True or False: Suppose we compile a program with 512-bit canaries, and the program produces no output (so it is impossible to leak the value of the canary). It is possible to successfully write to memory located above the stack canary.

◯ True ◯ False

Q1.2 True or False: In the last question of Project 1, ASLR prevents the attacker from knowing the address of any instructions in memory.

◯ True ◯ False

Q1.3 True or False: An 8-byte stack canary is less secure than a 4-byte stack canary.

◯ True ◯ False

Q1.4 Format string vulnerabilities can allow the attacker to:

☐ Read memory ☐ Execute Shellcode

☐ Write memory ☐ None of these

Q1.5 Which of the following memory safety hardening measures work by ensuring that all writeable regions in memory are non-executable, and all executable regions in memory are non-writeable?

☐ ASLR ☐ DEP (also known as W^X or NX)

☐ Stack canaries ☐ None of these

Q1.6 Bear Systems hardens its code with both DEP (also known as W^X or NX) and its own custom variant of ASLR. Normally, ASLR chooses a random offset for the stack and heap when the program starts running. Bear Systems modifies the compiler to choose a random offset when the program is compiled and hardcode this into the binary executable. Bear Systems ships the same executable to all of its customers. What is the effect of this modification to ASLR on security against memory safety exploits?

◯ This modification makes security better.

◯ This modification has no significant effect on security.

◯ This modification makes security worse.

Question 2 ***Virtual Tables, Real Fun***

The following code runs on a 32-bit x86 system.

```
#include <stdio.h>
int main() {
    FILE *fp;
    char buf[8];
    fp = fopen("outis", "rb");
    fread(buf, sizeof char, 12, fp);
    fclose(fp);
}
```

Behind the hood, the `FILE` struct is implemented in `stdio.h` as follows:

```
struct _IO_FILE; /* implementation omitted */

typedef struct {
    struct _IO_FILE ufile;
    struct _IO_jump_t *vtable;
} FILE;

struct _IO_jump_t {
    size_t (*fread)(void *, size_t, size_t, FILE *);
    size_t (*fwrite)(void *, size_t, size_t, FILE *);
    int (*fclose)(FILE *);
    /* more members below omitted */
};

int fclose(FILE *fp) { return fp->vtable->fclose(fp); }
/* more implementations below omitted */
```

Make the following assumptions:

1. No memory safety defenses are enabled.
2. The compiler does not perform any optimizations, reorder any variables, nor add any padding in between `struct` members.
3. The implementation of the function `fopen` has been omitted. Assume a sensible implementation of `fopen` that initializes the `ufile` and `vtable` fields of the `FILE` struct to sensible values.

Q2.1 Running the program in `gdb` using `invoke -d` as in Project 1, you find the following:

- `&buf = 0xbf608040`
- `&fp = 0xbf608048`
- `sizeof(struct _IO_FILE) = 32`

You wish to prove you can exploit the program by having it jump to the memory address `0xdeadbeef`. Complete the Python script below so that its output would successfully exploit the program.

NOTE: The syntax `\xRS` indicates a byte with hex value `0xRS`.

```
#!/usr/bin/env python2
import sys
sys.stdout.write('\x_____\x_____\x_____\x_____' + \
                 '\x_____\x_____\x_____\x_____' + \

                 '\x_____\x_____\x_____\x_____')
```

Q2.2 Which of the following defenses would stop your attack in part (a) from exploiting the program by jumping to memory address `0xdeadbeef`? Assume `0xdeadbeef` is at a read-only part of memory.

- ☐ Stack canaries
- ☐ W^X
- ☐ ASLR which does not randomize the `.text` segment (as in Project 1)
- ☐ ASLR which also randomizes the `.text` segment

Q2.3 (Consider this question independently from the previous part.) Now consider that we move the variables `fp` and `buf` outside of the `main` function, as follows:

```c
#include <stdio.h>
char buf[8]; /* &buf = 0x08402020 */
FILE *fp;    /* &fp  = 0x08402028 */
int main() { /* rest of main is the same, but no variables */ }
```

TRUE or FALSE: It is possible to modify the exploit in part (a) to exploit this modified program.

- ○ TRUE
- ○ FALSE