

Q1 Memory Safety: One Step backwards, Two Steps Forward 🧑🧑

(12 points)

Consider the following vulnerable C code:

```
1 void vulnerable(char *buf) {  
2     fread(buf - 5, 1, 29, stdin);  
3 }  
4  
5 int main() {  
6     char buf[8];  
7     vulnerable(buf);  
8     return 0;  
9 }
```

Stack at Line 2

RIP of (1)
SFP of (1)
canary of (1)
(2)
arg to (3)
RIP of (3)
SFP of (3)
canary of (3)

Assumptions:

- Stack canaries are enabled and consist of non-null bytes.
- All other memory safety defenses are disabled.
- You run GDB once and find that the RIP of `vulnerable` is located at `0xffffdc90`.
- You also find that the value of the RIP is `0x08ab04a9`.
- There is a `ret` gadget at address `0xf7ab04a9`.
- Your goal is to place and execute a 20-byte `SHELLCODE`.

Q1.1 (1 point) What goes in the blanks (1) - (3) in the stack diagram above?

Blank (1):	☐ <code>main</code>	☐ <code>vulnerable</code>	☐ <code>buf</code>
Blank (2):	☐ <code>main</code>	☐ <code>vulnerable</code>	☐ <code>buf</code>
Blank (3):	☐ <code>main</code>	☐ <code>vulnerable</code>	☐ <code>buf</code>

Q1.2 (1 point) What is the **address** of `buf - 5`?

☐ <code>0xffffdc88</code>	☐ <code>0xffffdc93</code>	☐ <code>0xffffdc98</code>
☐ <code>0xffffdc8f</code>	☐ <code>0xffffdc94</code>	☐ None of the above

Q1.3 (6 points) Provide an input to `fread` on line 2 that causes the program to execute `SHELLCODE`. If a part of the input can be any non-zero value, use `'A' * n` to represent `n` bytes of garbage.

(Question 1 continued...)

Consider Q1.4–Q1.5 independently of each other.

Q1.4 (1 point) Would the correct exploit (without modification) still succeed if Non-Executable Pages was enabled?

- ☐ Yes, because the **ret** gadget sits in executable memory and runs before reaching **SHELLCODE**.
- ☐ Yes, because Non-Executable Pages only restricts writes to the stack, not later execution.
- ☐ No, because **SHELLCODE** sits on the stack and the CPU will not execute non-executable pages.
- ☐ No, because the **ret** gadget lies in a non-executable region, so its execution faults first.

Q1.5 (1 point) Would the correct exploit (without modification) still succeed if ASLR was enabled?

- ☐ Yes, because the single GDB run reveals both the stack and **ret** gadget addresses.
- ☐ Yes, because the one-byte MSB overwrite does not depend on any randomized address.
- ☐ No, because ASLR randomizes the canary value, so the one-byte overwrite fails.
- ☐ No, because the stack address of **SHELLCODE** and the **ret** gadget change between runs.

Q1.6 (2 points) When does the correct exploit start executing instructions in **SHELLCODE**?

- ☐ Immediately after **fread** on line 2 returns
- ☐ Immediately after **vulnerable** on line 1 returns
- ☐ Immediately after the **ret** gadget at **0xf7ab04a9** executes
- ☐ Immediately after **main** on line 5 returns

Q2 Memory Safety: EvanBond, Double-O \x90

(19 points)

Consider the following vulnerable C code:

```
1 void q(int spectre) {
2     char goldfinger[8];
3     fread(goldfinger, 13, 1, stdin);
4     for (int i = 0; i < 8; i++) {
5         goldfinger[i] = 0x90;
6     }
7 }
8
9 void m() {
10     q(007);
11 }
```

Stack at Line 2

RIP of m
(1)
(2)
(3)
SFP of q
(4)

Assumptions:

- All memory safety defenses are disabled.
- You run GDB and break before executing line 3. You find that the RIP of **q** has the value **0xffffd6c0**.
- You also find that **goldfinger** is located at address **0xffffd620**.
- Recall that the byte **0x90** is also known as the NOP (no-operation) instruction.
- Your goal is to execute the 4-byte-long **SHELLCODE**.

Q2.1 (2 points) What values go in blanks (1) through (4) in the stack diagram above?

- | | | | |
|------------------------------------|---|--|---|
| ☐ (1) SFP of m | ☐ (2) RIP of q | ☐ (3) spectre | ☐ (4) goldfinger |
| ☐ (1) SFP of m | ☐ (2) spectre | ☐ (3) RIP of q | ☐ (4) goldfinger |
| ☐ (1) RIP of q | ☐ (2) spectre | ☐ (3) SFP of m | ☐ (4) goldfinger |
| ☐ (1) RIP of q | ☐ (2) goldfinger | ☐ (3) spectre | ☐ (4) SFP of m |

Q2.2 (4 points) Which of these values does the exploit have to overwrite to execute **SHELLCODE**? Select all that apply.

- | | |
|---|---|
| ☐ goldfinger | ☐ SFP of m |
| ☐ Least significant byte of the RIP of q | ☐ spectre |
| ☐ SFP of q | ☐ None of the above |

Q2.3 (4 points) Provide an input to the **fread** on Line 3.

If a part of the input can be any non-zero value, use **'A' * n** to represent **n** bytes of garbage.

(Question 2 continued...)

Q2.4 (2 points) Which memory safety defenses would cause the correct exploit from Q7.3 (without modifications) to fail? Consider each choice independently.

- ☐ Stack canaries ☐ None of the above
☐ Non-executable pages

Q2.5 (3 points) Which **values** of the RIP of **q** would cause the correct exploit from Q7.3 (without modifications) to fail? Select all that apply.

- | | | |
|-------------------------------------|-------------------------------------|---|
| ☐ 0xffffd640 | ☐ 0xffffd610 | ☐ 0xffffdbd0 |
| ☐ 0xffffd630 | ☐ 0xffffd600 | ☐ 0xffffdbc0 |
| ☐ 0xffffd620 | ☐ 0xffffdbe0 | ☐ None of the above |

Q2.6 (4 points) When does the correct exploit from Q7.3 start executing instructions in **SHELLCODE**?

- ☐ When the function **fread** on line 3 returns
☐ When the **for** loop on line 4 completes
☐ When the function **q** on line 1 returns
☐ When the function **m** on line 8 returns