

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): | ☒ main | ☐ vulnerable | ☐ buf |
| Blank (2): | ☐ main | ☐ vulnerable | ☒ buf |
| Blank (3): | ☐ main | ☒ vulnerable | ☐ buf |



(Question 1 continued...)

Solution: The stack diagram is as-follows:

Address	Content
0xffffdca8	[4] RIP of <code>main</code>
0xffffdca4	[4] SFP of <code>main</code>
0xffffdca0	[4] canary of <code>main</code>
0xffffdc98	[8] <code>buf</code>
0xffffdc94	[4] <code>*buf</code>
0xffffdc90	[4] RIP of <code>vulnerable</code>
0xffffdc8c	[4] SFP of <code>vulnerable</code>
0xffffdc88	[4] canary of <code>vulnerable</code>

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

- ☐ 0xffffdc88 ☒ 0xffffdc93 ☐ 0xffffdc98
☐ 0xffffdc8f ☐ 0xffffdc94 ☐ None of the above

Solution: `buf` starts at 0xffffdc98, so `buf - 5 = 0xffffdc93`. This address is the MSB of the 4-byte RIP of `vulnerable` stored little-endian at [0xffffdc90..0xffffdc93]. The 4-byte slot immediately above RIP of `vulnerable` is the pointer argument that `main` pushed before `call vulnerable`, occupying [0xffffdc94..0xffffdc97]; `buf[0]` sits at 0xffffdc98, directly above that argument slot.

(Question 1 continued...)

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.

`'\xf7' + '\x98\xdc\xff\xff' + SHELLCODE + 'A' * 4`

Solution: The one-byte write at `buf - 5 = 0xffffdc93` overwrites the MSB of RIP of `vulnerable`, redirecting it to the `ret` gadget at `0xf7ab04a9`.

When `vulnerable` executes `ret`, it pops the overwritten RIP into EIP (transferring control to the gadget) and advances ESP to `0xffffdc94` — the pointer-argument slot. The gadget is itself a `ret`, which pops the next 4 bytes (the argument slot) into EIP. So those 4 bytes must hold the shellcode address.

Place `SHELLCODE` starting at `buf[0] = 0xffffdc98`. Its 20 bytes span `[0xffffdc98..0xffffdcab]`, overwriting `buf`, canary of `main`, SFP of `main`, and RIP of `main`. This is safe: control is hijacked in `vulnerable`'s epilogue, so `main` never reaches its own epilogue and its canary is never checked. Canary of `vulnerable` at `0xffffdc88` is untouched since the write starts at `0xffffdc93` and runs upward.

Layout of the 29-byte input starting at `buf - 5 = 0xffffdc93`:

- byte 0: `\xf7` (MSB overwrite of RIP of `vulnerable`)
- bytes 1–4: `\x98\xdc\xff\xff` (shellcode address, into arg slot)
- bytes 5–24: 20-byte `SHELLCODE` (at `buf[0]` onward)
- bytes 25–28: 4 bytes of trailing filler

Input string: `'\xf7' + '\x98\xdc\xff\xff' + SHELLCODE + 'A' * 4`

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.

Solution: The `ret` gadget is in the text segment and executes fine. The problem is `SHELLCODE` sits on the stack. With NX enabled, the CPU refuses to execute instructions from stack pages, so when the gadget's `ret` transfers control to the shellcode address, a fault occurs.

(Question 1 continued...)

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.

Solution: ASLR randomizes the stack base and library load addresses per run. The GDB address `0xffffdc90` observed once does not persist, and the `ret` gadget's address also shifts. Both the shellcode target and the gadget become unknown.

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

Solution: The exploit chains two returns. When `vulnerable`'s epilogue runs, `ret` pops the overwritten RIP of `vulnerable` — now `0xf7ab04a9` — into EIP, transferring control to the `ret` gadget, and advances ESP to the argument slot at `0xffffdc94`. The gadget is itself a `ret` instruction, which pops the next 4 bytes (the argument slot, holding the shellcode address `0xffffdc98`) into EIP. Only then does control reach `SHELLCODE`.

`vulnerable` returning alone redirects to the gadget, not the shellcode. `fread` returning simply completes the write; execution continues into `vulnerable`'s epilogue. `main` never returns because control is hijacked during `vulnerable`'s epilogue.

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 |

Solution: The stack diagram:

0xffffd638	[4]	RIP of m
0xffffd634	[4]	SFP of m
0xffffd630	[4]	spectre
0xffffd62c	[4]	RIP of q
0xffffd628	[4]	SFP of q
0xffffd620	[8]	goldfinger

(Question 2 continued...)

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.

'A' * 8 + SHELLCODE + '\x20'

Solution: **fread** gives us 13 bytes to write, with which we will need to:

- overwrite **goldfinger** with 8 bytes of grab to reach the SFP of **q**
- Then we place our 4-byte **SHELLCODE**
- Finally we overwrite the LSB of the RIP of **q** so that we can point it somewhere into **goldfinger** or directly to the **SHELLCODE**. Any byte choice between **'\x20'** and **'\x28'** would have the RIP of **q** pointing either into **goldfinger** (which recall is a no-op sled thanks to the for loop beginning on line 4) or directly at **SHELLCODE**, which was placed at the address of the SFP of **q** (0xffffd628).

We cannot place **SHELLCODE** into **goldfinger**, as that same for loop would corrupt the bytes of **SHELLCODE**, leaving us with no choice but to place it right above **goldfinger**.

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 | |

Solution: Stack canaries would foil this exploit in two ways:

1. They would add 4 bytes onto the stack in between **goldfinger** and the SFP of **q**, meaning that we can no longer fully overwrite the SFP or overwrite the LSB of the RIP.
2. The intended exploit would also overwrite the value of the canary, causing the program to abort

Seeing as we are writing our exploit onto the stack, setting the stack to non-executable (through non-executable pages) would also result in us being unable to execute the instructions of **SHELLCODE**.

(Question 2 continued...)

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 |

Solution: Seeing as we can only overwrite the LSB of the RIP of **q**, we need the three most-significant bytes of the value the RIP to match the address of **goldfinger** (which is 0xffffd620). Thus any value of the RIP of **q** which is not in the form 0xffffd6__ would cause the program to fail.

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

Solution: The exploit works by overflowing **goldfinger** to overwrite RIP of **q** on the stack. This corrupted return address now points to the location of the **goldfinger** buffer (which the for loop conveniently turns into a NOP sled) leading to the instructions of **SHELLCODE** (which was placed just above **goldfinger**) to be executed.