

Q1 *Memory Safety: Jonah Dreams of Sheep* 🐑

(12 points)

Consider the following vulnerable C code:

```
1 void sleep() {
2     char dream[24];
3     fread(dream, 24, 1, stdin);
4     count_sheep(dream);
5 }
6
7 void count_sheep(char* input) {
8     size_t num_sheep;
9     fread(&num_sheep, sizeof(size_t), 1, stdin);
10
11     if (num_sheep > 64) {
12         exit(1);
13     }
14     printf(input);
15 }
```

Stack at Line 9

RIP of sleep
(1)
(2)
(3)
RIP of count_sheep
SFP of count_sheep
num_sheep

Assumptions:

- All memory safety defenses are disabled.
- There is a copy of `SHELLCODE` at address `0xdeadbeef`.
- There is no compiler padding.
- RIP of `sleep` is located at address `0xffffd634`.

Q1.1 (0.5 points) What goes in blank (1) in the stack diagram above?

- ☐ dream ☐ input ☐ num_sheep ☐ RIP of sleep ☒ SFP of sleep

Q1.2 (0.5 points) What goes in blank (2) in the stack diagram above?

- ☒ dream ☐ input ☐ num_sheep ☐ RIP of sleep ☐ SFP of sleep

Q1.3 (0.5 points) What goes in blank (3) in the stack diagram above?

- ☐ dream ☒ input ☐ num_sheep ☐ RIP of sleep ☐ SFP of sleep

Q1.4 (1.5 points) Which of the following memory safety vulnerabilities are present in the above code?

- ☐ Stack Buffer Overflow ☐ Off-by-one ☐ None of the above
☐ Heap Buffer Overflow ☒ Format String Vulnerability



Warning: Q1.5 and Q1.6 are very hard. Consider attempting all other questions before spending too much time on these two parts.

Q1.5 (5 points) Provide an input to the `fread` call on Line 3 that will help us execute `SHELLCODE`.

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

For the purposes of this question, the `%*u` specifier reads one argument from the stack (call it `m`), treats it as an integer, and prints `m` bytes. It then proceeds to read a second argument off the stack, but does nothing with it.

Hint: When processed by `printf`, `%hnn` writes one byte to the address given by the corresponding argument; the value written is the number of characters printed so far.

<code>\x0c\xd6\xff\xff</code>	<code>'</code>	<code>+ '%*u' +</code>	<code>(' %c' * </code>	2	<code>) +</code>
<code>'%hnn'</code>	<code>+</code>	<code>\xef\xbe\xad\xde</code>	<code>+</code>	<code>('A' * </code>	5

Solution:

Let's first take a look at the stack diagram:

0xffffd634	[4]	RIP of <code>sleep</code>	arg12
0xffffd630	[4]	SFP of <code>sleep</code>	arg11
0xffffd618	[24]	<code>dream</code>	arg5 - arg10
0xffffd614	[4]	<code>input</code>	arg4
0xffffd610	[4]	RIP of <code>count_sheep</code>	arg3
0xffffd60c	[4]	SFP of <code>count_sheep</code>	arg2
0xffffd608	[4]	<code>num_sheep</code>	arg1

Our exploit resembles the method for exploiting off-by-one vulnerabilities. The main idea of our exploit is to modify an SFP pointer to cause the stack to become “misaligned”, so that a location that already has `&SHELLCODE` stored in memory is interpreted by the CPU as a RIP.

Why so complicated? Why not just overwrite a RIP stored on the stack? Well, the format string vulnerability will allow us (the attacker) to write one byte of our choice somewhere on the stack. Overwriting just one byte of a RIP is not enough to replace it with the address of `SHELLCODE`. We don't have any way to write an entire word (4 bytes), because that would require a long format specifier (e.g., to write each of the bytes separately), and `dream` isn't big enough for that. So we need to get clever.

In more detail:

We note that the hint tells us to use a format specifier, and we take advantage of the unsanitized `printf` call on line 14 to execute a format string attack.

We will overwrite one byte of the SFP of `count_sheep`. We know that it contains the value `0xffffd630` (the address of SFP of `sleep`). The format string vulnerability allows us to manipulate the

least significant byte of this value, with a `%hnn` format specifier. We can store the address of `SHELLCODE` somewhere in our `dream` buffer and, using the `%hnn` format specifier, rewrite the SFP of `count_sheep` to point 4 bytes below the address where `&SHELLCODE` is stored.

To do this, we need to make sure that `%hnn` writes to the address where the SFP of `count_sheep` is stored, i.e., to `0xffffd60c`. So, we write `0xffffd60c` into the start of `dream` (the position marked `arg5`, i.e., the 5th argument that `printf` reads from), and we'll construct the format string so that the `%hnn` is the 5th specifier in the format string.

The first format specifier in the string will be `">%u'`. This will read two words from the locations marked `arg1`, `arg2`, i.e., from `num_sheep` and the SFP of `count_sheep`, and write `num_sheep` many bytes. Since we can control the contents of `num_sheep` (by specifying an appropriate input to the `fread` on line 9), we control how many bytes `printf` writes. This will be useful in a moment.

Next, we need to consume the 3rd, and 4th argument to `printf` (`arg3`, `arg4`) and effectively ignore them. We use 2 `%c` format specifiers for this purpose, so that our argument pointer moves up to point to the start of `dream` (`arg5`).

Next, we add `%hnn'`. This format specifier will read an address from `arg5` (the start of `dream`) and write one byte to that address. We've previously arranged that this address will be `0xffffd60c`, i.e., the address where the SFP of `count_sheep` is stored. So this will overwrite the first byte, i.e., the least significant byte, of the SFP of `count_sheep` (with a value that the attacker can control, by controlling the input to the `fread` on line 9).

Then, we add the address `\xef\xbe\xad\xde`. This 4-byte value needs to be stored somewhere on the stack, and here is a good spot. By overwriting the SFP of `count_sheep`, we'll cause the stack to be mis-aligned so that this location on the stack gets treated by the CPU as the RIP of `sleep`. `\xef\xbe\xad\xde` appears 15 bytes after the beginning of `dream`, i.e., at address `0xffffd627`. Therefore, we want to mis-align the stack so that address `0xffffd623` is treated as the SFP of `sleep` and address `0xffffd627` is treated as the RIP of `sleep`. We can do this by overwriting the SFP of `count_sheep` so it contains the address `0xffffd623`, i.e., overwriting its least significant byte with `0x23`.

To overwrite its least significant byte with `0x23`, we need to arrange that `printf` has printed `0x23` (i.e., 35 in decimal) bytes. Before the `%hnn` specifier, the format string prints `\x0c\xde\xff\xff` (4 bytes), whatever is printed by the `%*u` (`num_sheep` bytes), then whatever is printed by `%c` * 2 (2 bytes). Therefore, we want `num_sheep` + 6 to be 35, which requires `num_sheep` = 29 = `0x1D`.

When `count_sheep` returns, the prologue of `count_sheep` will execute `pop %ebp`, which will set `%ebp` to `0xffffd623` (since the location storing the SFP of `count_sheep` was modified to hold `0xffffd623`). When `sleep` returns, its prologue will first execute `mov %ebp, %esp`, which copies the value in `%ebp` (`0xffffd623`) into `%esp` (now `%esp` contains `0xffffd623`); then executes `pop %ebp`, which adds 4 to `%esp` (now `%esp` contains `0xffffd627`); then executes `ret`, which reads the pointer stored at `0xffffd627` and starts executing code there. Since we previously arranged that `&SHELLCODE` is stored at address `0xffffd627`, the `ret` in `sleep`'s prologue will start executing `SHELLCODE`.

There are multiple valid answers to this question. `\xef\xbe\xad\xde` can be moved up to 3 bytes later in the exploit string, as long as the answer to Q11.6 is increased by the same amount. It can also be moved earlier (but not into the first four bytes), with adjustments to the answer to Q11.6.

(Question 1 continued...)

Q1.6 (2 points) Provide an input to the `fread` call on Line 9 that, together with your answer to the previous part, will execute `SHELLCODE`.

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

`'\x1D\x00\x00\x00'`

Q1.7 (1 point) Which memory safety defenses would cause the correct exploit (without modifications) to fail? Consider each choice independently.

☒ ASLR

☒ Non-Executable Pages

☐ None of the above

Solution: ASLR would prevent the exploit, because the RIP of `sleep` would be at a different address.

We had a bug in this question. We didn't clearly specify what memory region `SHELLCODE` is stored in. As a result, it's not possible to tell reliably whether non-executable pages would stop the exploit. If `SHELLCODE` is stored in the heap or stack, non-executable pages would stop the exploit; if for some reason `SHELLCODE` was found in the code region (e.g., it already happened to appear in the code!), then non-executable pages would not stop the exploit. Therefore, we will ignore whether or not you marked the 2nd option, and only grade the 1st option.

Q1.8 (1 point) Would the same exploit work if the call to `fread` on line 3 was replaced with `fgets` (with the same parameters)?

☒ Yes, because we can still write in the correct exploit input into `dream`.

☐ Yes, because `fgets` does the same thing as `fread` when given the same number of bytes to read.

☐ No, because `fgets` will overwrite the last byte of the SFP of `sleep` with a null terminator.

☐ No, because the amount of space in `dream` required to perform the exploit will no longer be sufficient.

Solution: The key difference between `fread` and `fgets` here is that given a buffer of `n` bytes, `fread` will read exactly `n` bytes into the buffer, while `fgets` will read `n-1` bytes into the buffer and append a null terminator. Since our exploit input is only 21 bytes long and does not contain any null bytes, it could be used with either `fread` or `fgets`.