Assignment 3

Task: Consider the following three code snippets written in C, each of which contains one or more memory-safety vulnerabilities. For eash snippet identify the vulnerability / ies, the line that contains it, and describe an input that would trigger this vulnerability. If possible, try to correct the snippet, so it would be safe.

a.

```
void fun1(size_t len, char *data) {
      char *buf = malloc(len+5);
      memcpy(buf, data, len);
      buf[len] = '!';
      buf[len+1] = '\n';
      buf[len+2] = '\0';
}
```

Vulnerabilities:

1. Potential NULL pointer dereference: malloc can fail, returning NULL, leading to undefined behavior when accessed.
2. Buffer overflow: `memcpy(buf, data, len)`; assumes that data has at least `len` bytes.
3. Writing out of bounds: If len is very large (close to `SIZE_MAX`), `len + 5` can overflow, leading to a small allocation.

Triggering Input:

If `len = SIZE_MAX - 4, malloc(len + 5)` overflows, causing a small allocation instead of the expected large buffer.

Fix:

```
void fun1(size_t len, char *data) {
    if (len > SIZE_MAX - 5) return;  // Prevent integer overflow
    char *buf = malloc(len + 5);
    if (!buf) return;  // Check allocation success

    memcpy(buf, data, len);
    buf[len] = '!';
    buf[len+1] = '\n';
    buf[len+2] = '\0';

       free(buf); // Avoid memory leak
   }
```

b.

```
void fun2(size_t len, char *data) {
      char *buf = malloc(len+80);
      memcpy(buf, data, len);
      printf(buf);
}
```

Vulnerabilities:

1. Format String Vulnerability: `printf(buf);` is unsafe because `buf` might contain format specifiers `(%s, %x, etc.)`, leading to security exploits.
2. Unchecked `malloc` Return: If `malloc` fails, accessing `buf` leads to undefined behavior.

Triggering Input:

Passing a string like `"User input: %s %s %s"` could leak memory content.

Fix:

```
void fun2(size_t len, char *data) {
    if (len > SIZE_MAX - 80) return;
    char *buf = malloc(len + 80);
    if (!buf) return;

    memcpy(buf, data, len);
    buf[len] = '\0'; // Ensure null termination

    printf("%s", buf); // Use format specifier to prevent
exploits

    free(buf);
}
```

c.

```
char str1[8];
char str2[11];
gets(str2);
strncpy(str1, str2, 8);
printf("100% success: str1(%s), str2(%s)\n", str1, str2);
```

Vulnerabilities:

1. `gets(str2);` is dangerous: `gets()` does not check buffer size, leading to buffer overflow.
2. `strncpy(str1, str2, 8);` issue: If `str2` is exactly 8 or more characters, `str1` might not be null-terminated.
3. Format string issue in `printf`: `100%` should be escaped as `100%%`.

Triggering Input:

Entering a string longer than 10 characters will overflow str2, corrupting adjacent memory.

Fix:

```
    char str1[8];
    char str2[11];

    if (!fgets(str2, sizeof(str2), stdin)) return 1; // Safe
input handling

    str2[strcspn(str2, "\n")] = '\0'; // Remove newline
    strncpy(str1, str2, sizeof(str1) - 1);
    str1[sizeof(str1) - 1] = '\0'; // Ensure null-termination

    printf("100%% success: str1(%s), str2(%s)\n", str1, str2);
    return 0;
```

d.

```
int fun3(char *buf1, char *buf2, unsigned int len1,
unsigned int len2){
      char mybuf[256];

      if ((len1+len2)>256){
            return -1;
      }

      memcpy(mybuf, buf1, len1);
      memcpy(mybuf + len2, buf2, len2);

      printf("%s\n", mybuf);

      return 0;}
```

Vulnerabilities:

1. Off-by-one error in buffer overflow check: If `len1 + len2 == 256`, the last byte is out of bounds.
2. Potential uninitialized memory read: If `buf2` is a string but `len2` is small, `printf("%s\n", mybuf);` might read uninitialized memory.
3. `memcpy(mybuf + len2, buf2, len2);` is incorrect: Should be mybuf + len1.

Triggering Input:

- `len1 = 200, len2 = 60` would cause an overflow.
- `len1 = 5, len2 = 5, buf2 = "ABCD\0"` could lead to garbage output.

Fix:

```
int fun3(char *buf1, char *buf2, unsigned int len1, unsigned int
len2) {
    char mybuf[256];

    if (len1 + len2 >= sizeof(mybuf)) {  // Fix off-by-one
        return -1;
    }

    memcpy(mybuf, buf1, len1);
    memcpy(mybuf + len1, buf2, len2); // Corrected indexing

    mybuf[len1 + len2] = '\0'; // Ensure null termination

    printf("%s\n", mybuf);
    return 0;
```

Summary of Issues & Fixes

Function	Vulnerability	Fix
`fun1`	Integer overflow, unchecked `malloc`, buffer overflow	Check `malloc`, limit size, free memory
`fun2`	Format string vulnerability, unchecked `malloc`	Use printf(`"%s", buf`), check malloc
`gets() & strncpy()`	Buffer overflow (`gets`), lack of null-termination	Use `fgets`, manually null-terminate
`fun3`	Buffer overflow, off-by-one, uninitialized memory	Correct length check, null-terminate buffer