Question 1:

a)

The C program has vulnerabilities in the following lines: 13, 14.

1. Line 13 --> gets(buffer);

The function "gets(buffer);" does not check the input length for the buffer allowing inputs which are larger

than 64 bits which will leads to a stack overflow. This can overwrite the return address, giving a possible attacker

an execution to the "secret_function()" where you can open "/bin/sh", the shell.

C Code:

```
char buffer[80]; -->input  over 64 bits
memset(buffer, 'X', 72);  --> fill the buffer with x's
memcpy(buffer + 72 , 'secret_function() address'); --> overwrite the return address
```

2. Line 14 --> print(buffer)

The function "print(buffer);" gives an attacker the possibly to access memory or overwrite values using "%s" or "%x".

C Code:

```
void vulnerable_function()
{
```

```
    char buffer[64];
    printf("Input");
    gets(buffer);
    printf("%x", buffer);
  }
```

b)

To fix the security issues we can do the following.

Frist of all we can use "fgets()" instead of "gets()". "fgets()" will always check the input from the user and not allow it to exceed the buffer size.

Second of all for the "print(buffer);" we can just add "printf("%s",buffer);", this will ensure that only a string will be printed not allowing for any modification by the attacker.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void secret_function() {
  printf("Congratulations! You've successfully exploited a buffer overflow!\n");
  system("/bin/sh");
}

void vulnerable_function() {
  char buffer[64];
  printf("Enter your input: ");
  fgets(buffer, sizeof(buffer), stdin);
```

```
    printf("%s", buffer);
}

int main() {
    printf("Welcome to the secure program!\n");
    vulnerable_function();
    printf("Exiting normally...\n");
    return 0;
}
```

Question 2:

a)

The C program has a vulnerability due to line 19.

In line 17, "free(ptr);" , the pointer ptr is freed but on line "17" the pointer is called again after the deletion.

This can cause a vulnerability for the program because after freeing the pointer the memory which was allocated for it can be used

by another part of the program.

b)

A fix for this issue would be to add a "ptr = NULL" after line 17 which will cause a crash if the pointer is used after the memory freeing.

C Code:

```
#include <stdio.h>
```

```
#include <stdlib.h>

int main(void) {
   // Allocate memory on the heap for an integer
   int *ptr = malloc(sizeof(int));
   if (ptr == NULL) {
      perror("Memory allocation failed");
      return 1;
   }

   // Initialize and use the allocated memory
   *ptr = 100;
   printf("Value before free: %d\n", *ptr);

   // Free the allocated memory
   free(ptr);
   ptr = NULL;

   printf("Value after free (dangling pointer): %d\n", *ptr); // This will cause a crash

   return 0;
}
```

Question 3:

a)

To use an SQL Injection Attack, Alice has to use an SQL command, which will delete her record of borrowing the book, as an input for the database.

Alice should add something like this: DELETE FROM Borrowings WHERE MemberName = 'Alice' AND BookTitle = 'Hacking 101' AND BorrowDate = '2023-09-01'.

b)

To use an SQL Injection Attack, Bob should use an SQL command , which will delete the table "Borrowings", as an input.

Bob should add something like this: DROP TABLE Borrowings.