For a handy guide on x86 and GDB, check out this GDB Cheatsheet.

**Question 1  Stack Diagram Practice**

Consider the following function.

```
void swap(int* num1, int* num2, int arr_local[]) {
    int temp = *num1;
    *num1 = *num2;
    arr_local[0] = *num1;
    *num2 = temp;
    arr_local[1] = *num2;
}
```

```
int main(void) {
    int x = 61;
    int y = 1;
    int arr[2];
    swap(&x, &y, arr);
}
```

1. Complete the diagram of the stack if the code were executed until a breakpoint set on line 3. Assume normal (non-malicious) program execution. You do not need to write the values on the stack, only the names. There are no extraneous boxes, and each box represents 4 bytes in memory. The bottom of the page represents the lower addresses.

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

2. Now, draw arrows on the stack diagram denoting where the ESP and EBP would point if the code were executed until a breakpoint set on line 3.
Question 2  Software Vulnerabilities

For the following code, assume an attacker can control the value of basket, n, and owner_name passed into search_basket.

This code contains several security vulnerabilities. Circle three such vulnerabilities in the code and briefly explain each of the three on the next page.

```c
struct cat {
    char name[64];
    char owner[64];
    int age;
};

/* Searches through a BASKET of cats of length N (N should be less than 32). Adopts all cats with age less than 12 (kittens). Adopted kittens have their owner name overwritten with OWNER_NAME. Returns the number of kittens adopted. */
size_t search_basket(struct cat *basket, int n, char *owner_name) {
    struct cat kittens[32];
    size_t num_kittens = 0;
    if (n > 32) return -1;
    for (size_t i = 0; i <= n; i++) {
        if (basket[i].age < 12) {
            /* Reassign the owner name. */
            strcpy(basket[i].owner, owner_name);
            /* Copy the kitten from the basket. */
            kittens[num_kittens] = basket[i];
            num_kittens++;
            /* Print helpful message. */
            printf("Adopting kitten: ");
            printf(basket[i].name);
            printf("\n");
        }
    }
    /* Adopt kittens. */
    adopt_kittens(kittens, num_kittens); // Implementation not shown
    return num_kittens;
}
```
1. Explanation:

2. Explanation:

3. Explanation:

Describe how an attacker could exploit these vulnerabilities to obtain a shell:

---

---
Question 3  Remus

This problem is Question 1 of Project 1 converted to a discussion question, with the intention of providing a foundation for completing Project 1. The question will also consist of a live GDB walkthrough, conducted by your TA. A video version of this walkthrough is available on https://cs161.org/.

Consider the following vulnerable code.

```c
#include <stdio.h>

void orbit()
{
    char buf[8];
    gets(buf);
}

int main()
{
    orbit();
    return 0;
}
```

1. Which line of code contains the memory safety vulnerability? Briefly explain this vulnerability.

2. Complete the stack diagram if the code were executed until a breakpoint set on line 6. Assume normal (non-malicious) program execution. You do not need to write the values on the stack, only the names. There are no extraneous boxes, and each box represents 4 bytes in memory. The bottom of the page represents the lower addresses.

3. Construct an input to `buf` that would result in a successful buffer overflow attack. Assume that `orbit`'s RIP is stored at 0xfffff8e0 and that you have a SHELLCODE script that you would like to execute. In addition, assume that there are 8 bytes of compiler padding.