

# USER

Welcome to heap1!

I put my data on the heap so it should be safe from any tampering.

Since my data isn't on the stack I'll even let you write whatever info you want to the heap, I already took care of using malloc for you.

Heap State:

```
+-----+-----+
[*] Address -> Heap Data
+-----+-----+
[*] 0x616c07b152b0 -> pico
+-----+-----+
[*] 0x616c07b152d0 -> bico
+-----+-----+
```

1. Print Heap: (print the current state of the heap)
2. Write to buffer: (write to your own personal block of data on the heap)
3. Print safe\_var: (I'll even let you look at my variable on the heap, I'm confident it can't be modified)
4. Print Flag: (Try to print the flag, good luck)
5. Exit

Enter your choice:

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

```
#define FLAGSIZE_MAX 64
// amount of memory allocated for input_data
#define INPUT_DATA_SIZE 5
// amount of memory allocated for safe_var
#define SAFE_VAR_SIZE 5
```

```
int num_allocs;
char *safe_var;
char *input_data;
```

```
void check_win() {
    if (!strcmp(safe_var, "pico")) {
        printf("\nYOU WIN\n");

        // Print flag
        char buf[FLAGSIZE_MAX];
```

```

FILE *fd = fopen("flag.txt", "r");
fgets(buf, FLAGSIZE_MAX, fd);
printf("%s\n", buf);
fflush(stdout);

exit(0);
} else {
    printf("Looks like everything is still secure!\n");
    printf("\nNo flage for you :(\n");
    fflush(stdout);
}
}

void print_menu() {
    printf("\n1. Print Heap:\t\t(print the current state of the heap)"
        "\n2. Write to buffer:\t\t(write to your own personal block of data "
        "on the heap)"
        "\n3. Print safe_var:\t\t(I'll even let you look at my variable on "
        "the heap, "
        "I'm confident it can't be modified)"
        "\n4. Print Flag:\t\t(Try to print the flag, good luck)"
        "\n5. Exit\n\nEnter your choice: ");
    fflush(stdout);
}

void init() {
    printf("\nWelcome to heap1!\n");
    printf(
        "I put my data on the heap so it should be safe from any tampering.\n");
    printf("Since my data isn't on the stack I'll even let you write whatever "
        "info you want to the heap, I already took care of using malloc for "
        "you.\n\n");
    fflush(stdout);
    input_data = malloc(INPUT_DATA_SIZE);
    strncpy(input_data, "pico", INPUT_DATA_SIZE);
    safe_var = malloc(SAFE_VAR_SIZE);
    strncpy(safe_var, "bico", SAFE_VAR_SIZE);
}

void write_buffer() {
    printf("Data for buffer: ");
    fflush(stdout);
    scanf("%s", input_data);
}

```

```

void print_heap() {
    printf("Heap State:\n");
    printf("+-----+-----+\n");
    printf("[*] Address -> Heap Data \n");
    printf("+-----+-----+\n");
    printf("[*] %p -> %s\n", input_data, input_data);
    printf("+-----+-----+\n");
    printf("[*] %p -> %s\n", safe_var, safe_var);
    printf("+-----+-----+\n");
    fflush(stdout);
}

```

```

int main(void) {

    // Setup
    init();
    print_heap();

    int choice;

    while (1) {
        print_menu();
        if (scanf("%d", &choice) != 1) exit(0);

        switch (choice) {
            case 1:
                // print heap
                print_heap();
                break;
            case 2:
                write_buffer();
                break;
            case 3:
                // print safe_var
                printf("\n\nTake a look at my variable: safe_var = %s\n\n",
                    safe_var);
                fflush(stdout);
                break;
            case 4:
                // Check for win condition
                check_win();
                break;
            case 5:
                // exit
                return 0;

```

```

default:
    printf("Invalid choice\n");
    fflush(stdout);
}
}
}

```

Find the length needed for the overflow to make the second value pico per the source code

## ASSISTANT

To make the second value "pico" per the source code, we need to overflow the first buffer and overwrite the second buffer.

The first buffer `input\_data` has a size of `INPUT\_DATA\_SIZE` which is 5. However, we also need to consider the null terminator character `\0` that is added at the end of the string. So, the actual size of the buffer is 5 bytes for the string "pico" and 1 byte for the null terminator.

The second buffer `safe\_var` is located immediately after the first buffer in memory. To overwrite the second buffer, we need to write more than 5 bytes to the first buffer.

The length needed for the overflow to make the second value "pico" is 5 (size of the first buffer) + 1 (null terminator) + 4 (size of the second buffer) = 10 bytes.

However, since we are writing a string, we need to consider the null terminator character `\0` that is added at the end of the string. So, the actual length needed for the overflow is 10 bytes + 1 byte for the null terminator = 11 bytes.

But we can't write 11 bytes to the first buffer because it will cause a segmentation fault. We can write 10 bytes to the first buffer and the null terminator will overwrite the first byte of the second buffer.

So, the length needed for the overflow to make the second value "pico" is 10 bytes. The string to write to the first buffer would be "picoAAAAA" where "AAAAA" are the extra bytes needed to overflow the first buffer and overwrite the second buffer.

## USER

```

+-----+-----+
[*] Address -> Heap Data
+-----+-----+
[*] 0x616c07b152b0 -> AAAAAAAAAAAAAAAAAAAApico
+-----+-----+
[*] 0x616c07b152d0 -> bico
+-----+-----+

```

Wouldn't you need to perform hexadecimal math to figure out the number of A's?

## ASSISTANT

To determine the number of A's needed to overflow the first buffer and overwrite the second buffer, we need to perform hexadecimal math.

The address of the first buffer is 0x616c07b152b0 and the address of the second buffer is 0x616c07b152d0.

To find the difference between the two addresses, we can subtract the address of the first buffer from the address of the second buffer:

$$0x616c07b152d0 - 0x616c07b152b0 = 0x20$$

This means that the second buffer is 0x20 (32 in decimal) bytes away from the first buffer.

Since the first buffer has a size of 5 bytes, we need to write more than 5 bytes to overflow the first buffer.

To overwrite the second buffer, we need to write 5 (size of the first buffer) + 32 (distance between the two buffers) - 1 (because we are writing a string and the null terminator will overwrite the first byte of the second buffer) = 36 bytes.

However, since we are writing a string, we need to consider the null terminator character `\0` that is added at the end of the string. So, the actual number of A's needed is 36 bytes - 1 byte for the null terminator = 35 bytes.

But we can't write 36 bytes to the first buffer because it will cause a segmentation fault. We can write 35 bytes to the first buffer and the null terminator will overwrite the first byte of the second buffer.

So, the number of A's needed to overflow the first buffer and overwrite the second buffer is 35 bytes.

## USER

So what could I use for the exploit

## ASSISTANT

Based on the analysis, you could use a string of 35 'A's followed by the string "pico" to overflow the first buffer and overwrite the second buffer.

The exploit string would be:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAApico
```

This string is 35 bytes long, which is enough to overflow the first buffer and overwrite the second buffer, and it ends with the string "pico" which is the desired value for the second buffer.

**USER**

```
Heap State:
+-----+-----+
[*] Address -> Heap Data
+-----+-----+
[*] 0x616c07b152b0 -> AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAApico
+-----+-----+
[*] 0x616c07b152d0 -> AAAAAAAAAAApico
+-----+-----+
```

The second value needs to be pico

**ASSISTANT**

It looks like the exploit string I provided earlier was not correct.

Let's try again. Since the second buffer is 0x20 (32 in decimal) bytes away from the first buffer, and the first buffer has a size of 5 bytes, we need to write 5 (size of the first buffer) + 32 (distance between the two buffers) - 1 (because we are writing a string and the null terminator will overwrite the first byte of the second buffer) = 36 bytes.

However, since we are writing a string, we need to consider the null terminator character '\0' that is added at the end of the string. So, the actual number of A's needed is 36 bytes - 1 byte for the null terminator = 35 bytes.

But it looks like we need to overwrite the second buffer with the string "pico", so we need to add 4 more bytes to the exploit string.

The correct exploit string would be:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAApico
```

This string is 40 bytes long, which is enough to overflow the first buffer and overwrite the second buffer with the string "pico".

**USER**

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAApico
```

**ASSISTANT**

The correct exploit string is indeed:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAApico
```

This string is 32 bytes of 'A's, followed by the string "pico". The 32 bytes of 'A's overflow the first buffer and overwrite the memory location of the second buffer, and the string "pico" overwrites the contents of the

second buffer.

Well done!