## USER

Welcome to heap0!
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
+-------------+----------------+
[*]   0x60c9f6e662b0  ->   pico
+-------------+----------------+
[*]   0x60c9f6e662d0  ->   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:

Here is a challenge above

Source code:

```
#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, "bico") != 0) {
```

```
        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(write to your own personal block of data "
           "on the heap)"
           "\n3. Print safe_var:\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 heap0!\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();
int rval = scanf("%d", &choice);
if (rval == EOF){
    exit(0);
}
        if (rval != 1) {
            //printf("Invalid input. Please enter a valid choice.\n");
            //fflush(stdout);
            // Clear input buffer
            //while (getchar() != '\n');
            //continue;
    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);
        }
    }
}
```

heap 0
Author: Abrxs, pr1or1tyQ

Description
Are overflows just a stack concern?

## ASSISTANT

This challenge is a classic example of a heap overflow vulnerability.

Identify Vulnerability:

The vulnerability is in the `write_buffer` function, where the program uses `scanf` to read a string into the `input_data` buffer without checking its length. This buffer has a fixed size of `INPUT_DATA_SIZE`, which is 5.

If the user inputs a string that is longer than 5 characters, it will overflow the buffer and potentially overwrite adjacent memory locations. In this case, the `safe_var` buffer is allocated next to the `input_data` buffer in memory, so a successful overflow could potentially modify the value of `safe_var`.

In addition, the program checks whether `safe_var` has been modified by comparing it to the string "bico" in the `check_win` function. If `safe_var` has been modified, the program prints the flag.

Exploit Vulnerability:

To exploit this vulnerability, we can input a string that is longer than 5 characters into the `input_data` buffer. For example, we can input the string "bicoX", where the "X" represents a few extra characters that will overflow the buffer.

When we do this, we will overwrite the value of `safe_var`, causing it to be different from "bico". This will trigger the win condition in the `check_win` function, and the program will print the flag.

Here is a step-by-step guide to exploiting the vulnerability:

* Choose option 2 to write to the buffer
* Input a string that is longer than 5 characters, such as "bicoX\n" where "X\n" represents a single character per newline character
* Choose option 4 to check for the win condition
* The program will print the flag

Fix Vulnerability:

To fix this vulnerability, we need to ensure that the input buffer is not overflowed. We can do this by checking the length of the input string before copying it into the `input_data` buffer.

Here is the corrected `write_buffer` function:

```c
void write_buffer() {
    printf("Data for buffer: ");
    fflush(stdout);

    // Read a maximum of INPUT_DATA_SIZE - 1 characters from the input
    fgets(input_data, INPUT_DATA_SIZE, stdin);

    // Remove the newline character at the end of the string
    input_data[strcspn(input_data, "\n")] = 0;
}
```

By using `fgets` instead of `scanf`, we can prevent the input buffer from overflowing and ensure that the program is more secure.

## USER

Wouldn't it need to be at least length of [ ] 0x60c9f6e662b0 -
[*] 0x60c9f6e662d0

## ASSISTANT

The input string should be long enough to overflow the buffer and reach the `safe_var` buffer, which is located at `0x60c9f6e662d0`.

The address of `input_data` is `0x60c9f6e662b0`, and its length is 5. To overflow the buffer and reach `safe_var`, the input string should be at least 16 bytes long (to fill the `input_data` buffer and the padding bytes) plus the length of `safe_var` (which is also 5).

However, since the `safe_var` buffer is not null-terminated, we need to fill it completely to overwrite its value. So, the minimum length of the input string would be:

`0x60c9f6e662b0` to `0x60c9f6e662d0` is a difference of 16 bytes, which includes the length of `input_data` (5 bytes) and some padding bytes (11 bytes). To fill `input_data` and overwrite `safe_var`, we need an additional 5 bytes.

So, the minimum length of the input string would be `5 (input_data) + 11 (padding) = 16` to fill the `input_data` buffer and reach `safe_var`, and then `5` more to fill `safe_var` and modify its value.

Therefore, the correct length of the input string would be at least `16 + 5 = 21` bytes, but to be safe, we can add a few more bytes to ensure that we have filled the `safe_var` buffer completely and modified its value.

So, the correct length of the input string would be at least `21` bytes, and it should be filled with a pattern that will overwrite the `safe_var` buffer and modify its value.

## USER

So what would be an example exploit

## ASSISTANT

Here is an example exploit string that would work in this scenario:

`AAAAA` (fill `input_data` buffer) `BBBBBBBBBBBBBBB` (fill padding bytes) `pico` (overwrite `safe_var` buffer)

However, since we need to ensure that the entire `safe_var` buffer is filled and its value is modified, and since `safe_var` is not null-terminated, we can add a few more bytes to the exploit string:

`AAAAA` (fill `input_data` buffer) `BBBBBBBBBBBBBBB` (fill padding bytes) `picopicopico` (overwrite `safe_var` buffer)

This exploit string should fill the `input_data` buffer, the padding bytes, and the `safe_var` buffer, and modify its value to something other than "bico".

Note: In a real-world scenario, the exploit string might need to be more sophisticated, taking into account the specific memory layout, the size of the buffers, and the padding bytes.

Also, as a side note, this type of vulnerability is known as a "Heap Corruption" or "Heap Overflow" vulnerability, and it can be used to execute arbitrary code on the target system.

Example of the exact exploit command:

`./heap0
1
Heap State:
+-------------+----------------+
[*] Address   ->   Heap Data
+-------------+----------------+
[*]   0x60c9f6e662b0  ->   pico
+-------------+----------------+
[*]   0x60c9f6e662d0  ->   bico
+-------------+----------------+
2
Data for buffer: AAAABBBBBBBBBBBBBBBpicopicopico`

After running this command, the program should print the flag.