

<https://play.picoctf.org/practice/challenge/438>

If you're simply looking for examples of heap exploitation, you've come to the right place! This challenge provides a straightforward and effective demonstration of the concept.

**What is Heap Exploitation?**

Heap exploitation involves modifying the heap memory, typically one byte at a time. In computer programs, data is stored both on the stack and in the heap. The stack is used for data whose size is known at compile time, while the heap is used for dynamically allocated data, whose size may not be determined until runtime.

In C, the stack holds local variables, while the heap is typically populated through functions like malloc that allocate memory dynamically.

Programmers must ensure that functions don't accidentally write to or read from memory outside of the intended areas. If user input can be used to overwrite or read unintended parts of the heap, this becomes a heap exploitation scenario.

**The Challenge**

We are given access to a terminal through Netcat (using the nc command). On Linux, you can simply paste the provided command into a terminal. For Windows users, it can be installed via choco.

Netcat establishes a direct input-output connection over an internet port, allowing us to interact with the program through a text interface. Upon connecting, we see the contents of the heap and are presented with a set of commands. When attempting to print the flag, we are met with a message indicating that it's not so easy to access the flag:

**How to Proceed?**

By examining the source code, we quickly discover that the flag is stored in a file and can only be accessed by modifying the safe\_var variable, which is initially set to "bico".

void check\_win() {

if (strcmp(safe\_var, "bico") != 0) {

printf("\nYOU WIN\n");

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 flag for you :(\n");

fflush(stdout);

}

}

We also observe how safe\_var is initialized in the init() function:

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);

}

The author provides a small portion of the heap where we are free to write data. This is done using the write\_buffer() function:

void write\_buffer() {

printf("Data for buffer: ");

fflush(stdout);

scanf("%s", input\_data);

}

Here, the scanf function is key to solving the challenge. There are no boundary checks in place, so the input data will be written to memory until a whitespace character is encountered. As long as the heap location we want to corrupt is after the pointer's position, we can overflow the buffer and overwrite the "safe\_var" variable.

By using the "print heap" feature, we discover that the location where we can write (input\_data) is at 0x63c3882552b0, and the target variable (safe\_var) is at 0x63c3882552d0. The difference between these two addresses is 0x20 or 32 bytes.

This means that a string of 33 bytes (without spaces) will be enough to overwrite the safe\_var variable, granting us access to the flag.

**For Those on the Other Side of the Screen**

If you're developing server software in C that accepts direct user input, how can you protect your program from such exploits?

First off, kudos for taking on such a challenge.

To safeguard your program, ensure you're using functions that perform boundary checks when handling input whose size cannot be predicted. For example, use fgets instead of scanf:

char buffer[256];

fgets(buffer, sizeof(buffer), stdin);

This ensures that input doesn't overflow the buffer and helps prevent memory corruption.

