# Prog1

The first thing we have done has been to compile the program and to debug it, to know how the variables are located in memory (although the memory space that is assigned varies, these are always ordered in a certain way).

So we can say to illustrate this explanation that the memory space allocated for each variable is:

char buffer[8] --> 8 bytes in memory location 0x4028

int val1 --> 4 bytes at memory location 0x404030

int val2 --> 4 bytes at memory location 0x404034

Knowing that C does not have a certain endianess, but adapts to the characteristics of the machine on which it is executed, it has been verified that on the machine on which the program has been executed, the endianess is little endian. Therefore, buffer will be the variable written first to memory, then val1, val2 being the last.

Explanation of the code:

Texto

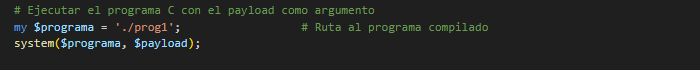
Descripción generada automáticamente

We enable strict and warnings to warn us of possible errors when executing the script.

Texto

Descripción generada automáticamente

We assign the values of val1 and val2 from the first iteration. Then the payload for the buffer overflow is constructed. A string consisting of 8 characters 'A' is created to fill the buffer. Then the values of val1 and val2 are added in little endian format using Perl's pack function.



We execute the C program prog1 with $payload as argument with the system function. In such a way that the payload is received as argument in the program and the buffer overflow occurs.

Texto

Descripción generada automáticamente

The rest of the code is to repeat the previously commented code, only that the values of val1 and val2 will be changed.

# Prog2

We repeat the process of prog1: compile, enter debugging and get the memory space allocated to each variable:  
Texto

Descripción generada automáticamente

You can see that the user variable is below the task variable. In this way, through a buffer overflow of the user variable, we can delete what was already stored in the task variable.

We write a simple script that would look like this:Captura de pantalla de un celular

Descripción generada automáticamente

We create a payload containing 10 As, so that we overexceed the size of the buffer allocated to char user[10], so that we would leave the memory space allocated to task[] at 0.

Texto

Descripción generada automáticamente

We finally got this as an output with only 10 As.

# Prog3

# Hello.c

This exercise could not be finished, however I will leave you the approach I followed so that you can understand the problem I had.

At the beginning of the exercise we are asked to write a harmless minimal C program (named hello.c) with a BOF vulnerability.

The program that I have written with these characteristics is the following one:  
Texto

Descripción generada automáticamente

In the we have two functions; in the first one we declare a buffer of characters called buffer with capacity for 8 characters. Then, the strcpy() function is used to copy the contents of the input argument into the buffer. It should be noted that we are not doing any buffer checking, so we could cause a buffer overflow.

In the main function (main), a character array called input is declared with a capacity of 16 characters. Then, a message is displayed on the screen asking the user to enter an input. The unsafe gets() function is used to read the user's input and store it in input. gets() does not perform any buffer size check and may lead to a buffer overflow if the user enters more than 16 characters.

The vulnerableFunction() function is then called passing input as an argument. This can lead to a buffer overflow inside vulnerableFunction() if the content of input is too long.

Let's compile it to get into debugging.

We open the command terminal and enter the following command to debug hello.c

Next we proceed to debug the program and set breakpointsTexto

Descripción generada automáticamente

Actually the only break that needs to be analyzed is in vulnerableFunction.

We start the debugging and we look at the value of rip, and the content of bufer and its position in memoryTexto

Descripción generada automáticamente

We can see that rip is at 0x7fffffffffffde18, the buffer is at position 0x7fffffffffffde08 and contains AAAAAAAAAA.

If we subtract the two positions we would have a space of 16 Bytes between them.

Note also the saved rip above, whose address is 0x5555555555555100.

We proceed to inject the entry to exploit the vulnerability:

AAAAAAAAAAAAAAAA\x00\x51\x55\x55\x55\x55\x00Hello, worldTexto

Descripción generada automáticamente

As we can see in the buffer is stored "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA," and gives us a Segmentation fail error.

The thing is that I have been trying this input with different lengths of A, and it always ends up giving me the same thing, so I don't really know what we are doing wrong.Texto

Descripción generada automáticamente

# Offbyone.c

In this program we have had similar failures to Hello.c