Sistemas de Software Seguros Segurança de Software

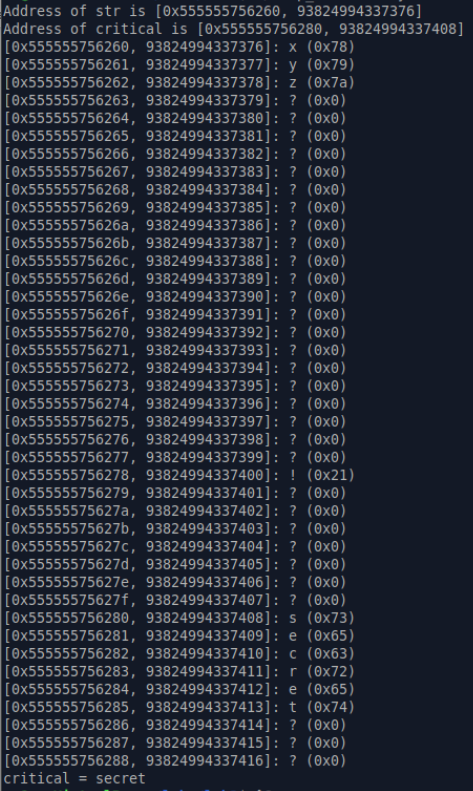
**2024/2025**

**Class Project: Experiments with Buffer Overflows**

# Heap Overflow

1. Understand the output from the above execution and relate it with the code you wrote in point a).

**R:** O output gerado foi o seguinte:



As primeiras duas linhas do output dão print ao enderenço de memória onde foram guardadas os ponteiros str e critical, respetivamente. Esse valor é mostrado em valor hexadecimal bem como num valor decimal, ou seja, a linha:

Address of str is [0x555555756260, 93824994337376]

Significa que o ponteiro str está guardado no endereço 0x555555756260 (que em decimal seria 93824994337376). Estas duas linhas são geradas pelas seguintes linhas de código:

printf("Address of str is [%p, %lu]\n", str, (unsigned long) str);

printf("Address of critical is [%p, %lu]\n", critical, (unsigned long) critical);

As quarenta e uma linhas seguintes mostram o mesmo que as linhas anteriores, mas em relação a cada posição de memória do ponteiro tmp, que vai desde o início da primeira posição de memória do ponteiro str até à última posição do ponteiro critical e, para além disso, mostra também o valor de forma “normal” (caso ainda não exista nenhum valor aparece “?”) e em hexadecimal para o qual o ponteiro tmp aponta . Como podemos observar na imagem acima muito dos endereços de memória ainda estão livres, como mostra o exemplo abaixo:

[0x555555756270, 93824994337392]: ? (0x0)

Cada linha destas é gerada pela seguinte linha de código (que está dentro dum loop while – daí ser impressa várias vezes):

printf("[%p, %lu]: %c (0x%x)\n", tmp, (unsigned long) tmp, isprint(\*tmp) ? \*tmp : '?', (unsigned) (\*tmp));

Por fim, a última linha dá o print do que está guardado no ponteiro critical. Nesta execução do programa o valor era secret. Esta linha aparece devido ao seguinte print:

printf("critical = %s\n", critical);

1. Run the program again, but now in such a way as to create an overflow that makes the printf of variable critical present the value “CIENCIAS” (without the quotes).

Como é possível observar no código existe uma vulnerabilidade devido ao strcpy(str, argv[1]), com isto, como não é especificado nem delimitado o número de caracteres que podemos inserir no terminal e o ponteiro str só tem quatro posições na memória exclusivas para ele próprio então, ao escrever um input no terminal maior que três caracteres (o último terá de ser o /0) vamos começar a ocupar outras posições de memória que podem já está a ser usadas para outras variáveis. Ao entender isto, para fazer com que a variável critical apresente o valor CIENCIAS, basta escrever uma string suficientemente grande para conseguirmos alcançar as posições de memória onde está cada letra desta variável.

O valor que inserimos no terminal foi o seguinte:  
  
 ./heap\_overflow xyzqweqweqweqweqweqweqweqweqweqwCIENCIAS

Com isto a última, tal como esperado deu o print “critical = CIENCIAS”.

# Stack Overflow

c) Look at the generated file (stack\_overflow.s) and determine the number of bytes needed to create an overflow of buffer buf and write over something relevant (like, RBP and RIP) in the stack. Justify.

10 – RBP – 16 ou 17 RIP

d) Confirm your result by compiling the code and executing it with the appropriate argument. Explain what has occurred.

* 1. Introduce the following C code in a file stack\_overflow\_2.c:

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

void test(char \*s)

{

char buf[16]; strcpy(buf, s);

}

void cannot()

{

printf("This function should not be executed! Was there a BO (-; ?\n"); exit(0);

}

int main (int argc, char\*\* argv)

{

printf(" &cannot = %p\n", &cannot); test(argv[1]);

printf("I’m OK!\n");

}

* 1. Compile the program in the following way:

gcc -fno-stack-protector -o stack\_2 stack\_overflow\_2.c

* 1. Run the program in the following way:

./stack\_2 12345

* 1. Confirm that the program did not print the following message: This function should not be executed! ... Justify.
  2. Introduce the following C code in a file call\_stack\_overflow\_2.c:

#include <malloc.h> #include <unistd.h>

int main()

{

int i, A, B, C, D, E, F, G, H, I;

char \*buf = (char \*) malloc(1000); char \*\*arr = (char \*\*) malloc(10);

A = 0;

B = 0;

C = 0;

D = 0;

E = 0;

F = 0;

G = 0;

H = 0;

I = 0;

for (i=0; i<A; i++) buf[i] = 'A';

buf[A] = B; buf[A+1] = C; buf[A+2] = D; buf[A+3] = E; buf[A+4] = F; buf[A+5] = G; buf[A+6] = H; buf[A+7] = I;

arr[0] = "./stack\_2"; arr[1] = buf;

arr[2] = 0x00; execv("./stack\_2", arr);

}

* 1. Substitute the values of A, B, C, D, E, F, G, H, I in the above code so that the program prints the message This function should not be executed!... Justify the values that you selected (Help: analyze like in question c)

# Delivery of the Report

The output of the class project is a report answering all the questions and including the justifications for the responses. Each group should deliver the report either by submitting it in the course Moodle page or, if there is some difficulty with this method, by emailing it to the TP professor. The file type should be a pdf. ***NOTE: only one element of the group needs to deliver the report!***

**Deadline**: 21 October 2024 (there will be no extensions)

======================================

**The following exercise is optional!**

======================================

# Challenge exercise

A server on the internet is vulnerable because it has a flaw in how it manages and accesses memory. Fortunately, the code implementing the server has been leaked, together with some client code that can interact with the server. I captured both codes and made them available on the course website, with the names serv-challenge.c and client-challenge.c.

You aim to collect in the client the value of a flag stored in the server, i.e., exploit the vulnerability to perform an information leak. The relevant server flag is **f14g**. How can you achieve this by only calling the server through the offered interface?

Some hints:

1. You cannot modify the server code, but you can change the client code as you like
2. Look through the server code and understand very well how each operation is implemented
3. What kind of vulnerability might the server have? It manages memory with malloc and free …
4. Remember that addresses have 8 bytes in an x86-64 architecture, and that information is stored in memory in a little-endian format
5. Probably, the vulnerability cannot be exploited by executing a single server operation. So, you will have to find an appropriate schedule of operations, with the correct inputs, to be able to collect your prize … flag **f14g**

Good luck!