Sistemas de Software Seguros Segurança de Software

**2024/2025**

**Class Project: Experiments with Buffer Overflows**

Gustavo Henrique Nº64361

Leonardo Monteiro Nº58250

Maria Figueirinhas Nº46494

# Heap Overflow

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

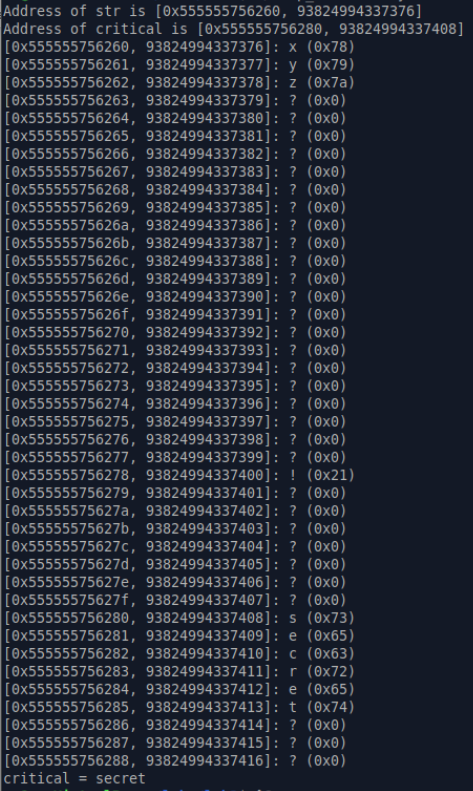
**R:** O output gerado foi o seguinte:

Figura - Tenho de rever isto

As primeiras duas linhas do output dão print ao enderenço de memória onde foram guardados 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).

**R:** 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.

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

c) e d) Combinadas.

**R:** Criámos um buffer buf[10], ou seja, um buffer que guarda na memória espaço para 10 bytes de informação. Como o strcpy() não tem em conta a necessidade do \0 no final, mantendo os 10 bytes, irá colocar \0 no final dos argumento que for fornecido, mesmo que este já tenha 10 bytes. Isto criará um *overflow*, sendo que o resultado tem 11 bytes e não 10 e começará a escrever por cima de RBP. Isto não terá problemas, em argumentos de tamanho igual ou inferior a 9 bytes.

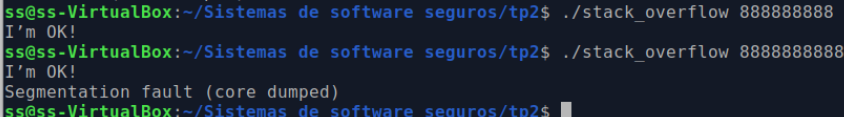


Figura 2 - Resultado quando o argumento fornecido tem apenas 9 bytes.

O resultado disto será uma *Segmentation Fault*, significando que o programa está a escrever num sítio que já não pertence ao programa em execução.

A blue text on a black background

Description automatically generated

Figura 3 - Resultado quando o argumento fornecido tem 10 bytes.

Uma vez que os argumentos escritos na função vão ser copiados por strcpy para o endereço -10RBP, então, para escrever por cima de RIP, teremos de fornecer um total de 18 bytes (10 para encher o buffer e 8 para encher o RBP). Com isto o \0 será o décimo nono byte, que irá ocupar a primeira posição do RIP.

h) Confirm that the program did not print the following message: This function should not be executed! ... Justify.

Uma imagem com texto, captura de ecrã, Tipo de letra, file

Descrição gerada automaticamente

Figura 5 - Print do terminal ao executar o ficheiro

**R:** Neste caso, o programa **não imprimiu** a mensagem "This function should not be executed!" porque **não houve um buffer overflow** que desviasse o fluxo de execução para a função cannot.

Ao executar o comando *./stack\_2 12345*, a string "12345" foi passada para a função test. A função test tenta copiar essa string para o buffer buf[16], que tem 16 bytes de tamanho. Como a string "12345" tem apenas 5 caracteres, não excede o tamanho do buffer, logo, **não ocorreu um buffer overflow.**

j) 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)

**R:** A função call\_stack vai fornecer os argumentos para a função stack\_2. As variáveis A, B, etc serão estes argumentos, como evidenciado pelas linhas:

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

arr[1] = buf;

arr[2] = 0x00;

execv("./stack\_2", arr);

O comando execv vai executar a função presente no primeiro argumento, (stack\_2), com os valores contidos em arr. Os valores contidos em ‘buf’ correspondem às variáveis inicializadas a 0 no código base (A-I). Uma vez que o objetivo é obter a mensagem ‘This function should not be executed!...’, teremos de recriar as condições nas quais a função stack\_2 produz este resultado.

Para tal, teremos de criar um buffer overflow de tamanho suficiente que preencha todo o espaço até chegar ao local na memória que guarda a chamada à função cannot. Chegámos à conclusão que esse valor teria de ser 24 (16 para preencher o buf e 8 para encher o RBP e, deste modo, começar a escrever no RIP, que é o espaço de retorno da função). Daremos esse valor de bytes a A. O ciclo for irá garantir que esse espaço ficará preenchido com a letra ‘A’. Quando atingirmos buf[A] (buf com índice igual ao valor de A), então podemos colocar o código que chama a string, usando o seu endereço (ver figura 4).

Como a stack se lê de cima para baixo, mas é escrita de baixo para cima, inicializamos B com a parte final do endereço necessário (little endian). Faremos o mesmo para todo o endereço, utilizando notação hexadecimal. Abaixo mostra-se a notação final.

A = 24;

B = 0x3c;

C = 0x47;

D = 0x55;

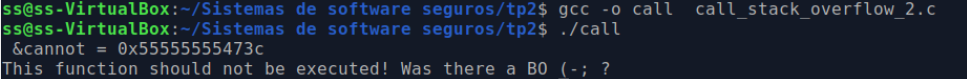
E = 0x55;

F = 0x55;

G = 0x55;

H = 0x00;

I = 0x00;

Com esta alteração, o resultado observado no terminal é o seguinte: