

Class Project: Experiments with Buffer Overflows

Gustavo Henrique N°64361

Leonardo Monteiro N°58250

Maria Figueirinhas N°46494

1. Heap Overflow

d) 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:

```
Address of str is [0x555555756260, 93824994337376]
Address of critical is [0x555555756280, 93824994337408]
[0x555555756260, 93824994337376]: x (0x78)
[0x555555756261, 93824994337377]: y (0x79)
[0x555555756262, 93824994337378]: z (0x7a)
[0x555555756263, 93824994337379]: ? (0x0)
[0x555555756264, 93824994337380]: ? (0x0)
[0x555555756265, 93824994337381]: ? (0x0)
[0x555555756266, 93824994337382]: ? (0x0)
[0x555555756267, 93824994337383]: ? (0x0)
[0x555555756268, 93824994337384]: ? (0x0)
[0x555555756269, 93824994337385]: ? (0x0)
[0x55555575626a, 93824994337386]: ? (0x0)
[0x55555575626b, 93824994337387]: ? (0x0)
[0x55555575626c, 93824994337388]: ? (0x0)
[0x55555575626d, 93824994337389]: ? (0x0)
[0x55555575626e, 93824994337390]: ? (0x0)
[0x55555575626f, 93824994337391]: ? (0x0)
[0x555555756270, 93824994337392]: ? (0x0)
[0x555555756271, 93824994337393]: ? (0x0)
[0x555555756272, 93824994337394]: ? (0x0)
[0x555555756273, 93824994337395]: ? (0x0)
[0x555555756274, 93824994337396]: ? (0x0)
[0x555555756275, 93824994337397]: ? (0x0)
[0x555555756276, 93824994337398]: ? (0x0)
[0x555555756277, 93824994337399]: ? (0x0)
[0x555555756278, 93824994337400]: ! (0x21)
[0x555555756279, 93824994337401]: ? (0x0)
[0x55555575627a, 93824994337402]: ? (0x0)
[0x55555575627b, 93824994337403]: ? (0x0)
[0x55555575627c, 93824994337404]: ? (0x0)
[0x55555575627d, 93824994337405]: ? (0x0)
[0x55555575627e, 93824994337406]: ? (0x0)
[0x55555575627f, 93824994337407]: ? (0x0)
[0x555555756280, 93824994337408]: s (0x73)
[0x555555756281, 93824994337409]: e (0x65)
[0x555555756282, 93824994337410]: c (0x63)
[0x555555756283, 93824994337411]: r (0x72)
[0x555555756284, 93824994337412]: e (0x65)
[0x555555756285, 93824994337413]: t (0x74)
[0x555555756286, 93824994337414]: ? (0x0)
[0x555555756287, 93824994337415]: ? (0x0)
```

Figura 1 – Executando o ficheiro heap_overflow com o input 'xyz'. Critical mantém-se com o valor correto – 'secret'.

As primeiras duas linhas do output dão print ao endereço de memória onde foram guardados os ponteiros *str* e *critical*, respectivamente. Esse valor é mostrado em valor hexadecimal e 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 é 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:

```
[0x55555756270, 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);
```

- e) Run the program again, but now in such a way as to create an overflow that makes the printf of variable criticalpresent 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 estar 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 xyzqwegwegwegwegwegwegwegwegwCIENCIAS
```

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

2. 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 passa a ter 11 *bytes* e não 10 e começará a escrever por cima de RBP. Contudo, não haverá problema, em argumentos de tamanho igual ou inferior a 9 *bytes*.

```
ss@ss-VirtualBox:~/Sistemas de software seguros/tp2$ ./stack_overflow 888888888
I'm OK!
```

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.

```
ss@ss-VirtualBox:~/Sistemas de software seguros/tp2$ ./stack_overflow 8888888888
I'm OK!
Segmentation fault (core dumped)
```

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.

```
ss@ss-VirtualBox:~/labs/lab2$ ./stack_2 12345
&cannot = 0x55555555473c
I'm OK!
```

Figura 4 - Print do terminal ao executar o ficheiro com um argumento adequado. '12345', neste caso.

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);
```

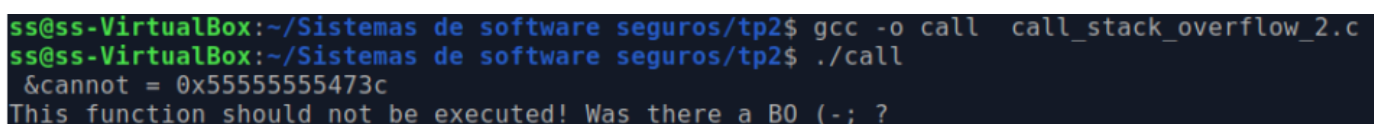
O comando *execv* vai executar a função presente no primeiro argumento, (*stack_2*), com os valores contidos em *arr*. Os valores contidos em '*buf*' correspondem às variáveis inicializadas a 0 no código base (A-I). Uma vez que o objetivo é obter a mensagem '*This function should not be executed!...*', 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:



```
ss@ss-VirtualBox:~/Sistemas de software seguros/tp2$ gcc -o call call_stack_overflow_2.c
ss@ss-VirtualBox:~/Sistemas de software seguros/tp2$ ./call
&cannot = 0x55555555473c
This function should not be executed! Was there a BO (-; ?
```

Figura 5 - Confirmamos que A = 24 preenche todo o espaço até ao local da função *cannot*.