

Universidade do Minho

MESTRADO EM ENGENHARIA INFORMÁTICA

**Engenharia de segurança**

**Ficha de exercício 11**

**Grupo 1**

RUI CARLOS AZEVEDO CARVALHO - PG47633

DANIEL BARBOSA MIRANDA - PG47123

ANA LUÍSA LIRA TOMÉ CARNEIRO - PG46983

# Buffer Overflow

## Pergunta P1.1 - Buffer overflow

Nesta pergunta era pretendido que explora-se-mos as vulnerabilidades de *buffer overflow* no programa 0-simple.c e no programa RootExploit.c de forma a obter as mensagens "YOU WIN!!!" e a mensagem "Foram-lhe atribuídas permissões de root/admin", respetivamente.

### 0-simple.c

No caso do programa 0-simple.c, este lê o *input* do utilizador para um *buffer* com espaço para 64 bytes. Existe ainda um variável 'control' que se encontra inicializada a 0 e que vai condicionar o *print* da mensagem "YOU WIN!!!". Caso a variável 'control' seja igual a 1 então o programa devolve a mensagem pretendida, se não o programa devolve a mensagem "Try again...". Como forma de alterarmos a variável 'control' para 1, podemos preencher o *buffer* que armazena o *input* do utilizador de forma a também preencher a variável 'control' com o valor 1 e assim conseguirmos obter a mensagem pretendida.

Como temos acesso ao código do programa podemos começar por determinar qual o endereço em memória do *buffer* e da variável 'control', calculando a diferença entre os endereços. Essa diferença será aproveitada para sabermos a quantidade de bytes que temos de preencher para conseguirmos chegar ao endereço da variável 'control'. Por exemplo, imagina-se que a diferença entre as variáveis era 76, então teríamos de enviar como *input* ao programa cerca de 76 inteiros para que o inteiro nº 77 fosse 1 e assim preencher o endereço da variável 'control' com o inteiro 1. Desta forma obteríamos a mensagem "YOU WIN!!!".

Devido a versão utilizada pelo gcc (gcc (Ubuntu 7.5.0-3ubuntu1 18.04) 7.5.0) não é possível colocar em prática os passos mencionados em cima, pois esta versão do compilador faz otimizações e implementa métodos que impedem certas falhas de segurança. Neste caso, o endereço da variável 'control' encontra-se num endereço menor que o endereço do *buffer*, impedido com que seja possível preencher o *buffer* de forma a também preencher a variável 'control'.

### RootExploit.c

Para conseguir resolver o problema para o caso do programa RootExploit.c, aplicam-se passos semelhantes aos apresentados acima. Este programa começa por pedir ao utilizador que insira uma password que será armazenada num *buffer* de 4 bytes. Existe ainda um variável 'pass' que se encontra inicializada a 0 e que vai condicionar o *print*

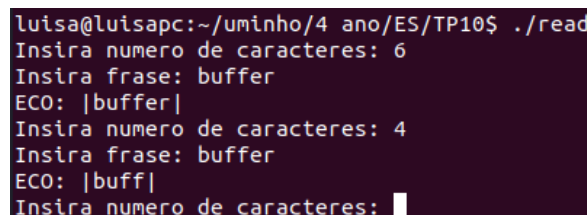
da mensagem "Foram-lhe atribuídas permissões de root/admin". Caso a variável 'pass' seja igual a 1 então o programa devolve a mensagem pretendida, se não o programa devolve unicamente a mensagem "Password errada". Uma das formas de alterar a variável 'pass' será em introduzir como input a password correta, contudo como não temos acesso a esta podemos preencher o *buffer* que armazena o *input* do utilizador de forma a também preencher a variável 'pass' com o valor 1 e assim conseguirmos obter a mensagem pretendida.

Como temos acesso ao código do programa podemos começar por determinar qual o endereço em memória do *buffer* e da variável 'pass', calculando a diferença entre os endereços. Essa diferença será aproveitada para sabermos a quantidade de bytes que temos de preencher para conseguirmos chegar ao endereço da variável 'pass'. Por exemplo, imagina-se que a diferença entre as variáveis era 10, então teríamos de enviar como *input* ao programa cerca de 10 inteiros para que o inteiro nº 11 fosse 1 e assim preencher o endereço da variável 'pass' com o inteiro 1. Desta forma obteríamos a mensagem "Foram-lhe atribuídas permissões de root/admin".

Devido a versão utilizada pelo gcc (gcc (Ubuntu 7.5.0-3ubuntu1 18.04) 7.5.0) não é possível colocar em prática os passos mencionados em cima, pois esta versão do compilador faz otimizações e implementa métodos que impedem certas falhas de segurança, tal como aconteceu com o programa 0-simple.c.

## Pergunta P1.2 - Read overflow

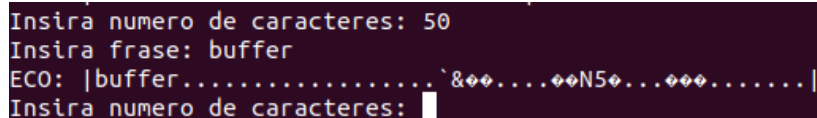
O programa ReadOverflow.c tem como objetivo ler uma mensagem e o respetivo número de caracteres enviada pelo utilizador, devolvendo no final a mensagem enviada ao programa. O funcionamento base do programa encontra-se na imagem abaixo.



```
luisa@luisapc:~/uminho/4 ano/ES/TP10$ ./read
Insira numero de caracteres: 6
Insira frase: buffer
ECO: |buffer|
Insira numero de caracteres: 4
Insira frase: buffer
ECO: |buff|
Insira numero de caracteres: █
```

Figure 1.1: Funcionamento base do programa ReadOverflow.c

Este programa começa por perguntar ao utilizador quantos caracteres é que deverão ser lidos pelo programa, perguntando logo de seguida que mensagem é que o utilizador pretende que o programa ler. Caso o número de caracteres a ler seja igual ao tamanho da mensagem então é retornada a mensagem completa e caso o número de caracteres a ler seja menor que o tamanho da mensagem então é retornada a mensagem com tamanho igual ao número de caracteres introduzidos. Contudo, o caso que provoca uma vulnerabilidade de read overflow acontece quando tentamos ler mais caracteres do que os existentes na mensagem tal como está na imagem abaixo.



```

Insira numero de caracteres: 50
Insira frase: buffer
ECO: |buffer.....&...N5...|
Insira numero de caracteres: 

```

Figure 1.2: Read Overflow no programa ReadOverflow.c

No caso apresentado podemos ver que não só é devolvida a mensagem introduzida como também é devolvida outra informação que esteja contida na memória do programa. Desta forma, é possível que um atacante tenha facilmente acesso a informação privada e sensível que esteja armazenada no programa bastando com que este só tenha de introduzir um número elevado de caracteres para obrigar o programa a ler esses caracteres da memória.

Uma das vulnerabilidades mais famosas encontrada na biblioteca de software de criptografia *open-source*, OpenSSL, foi o HeartBleed e baseava-se precisamente neste problema do read overflow. Uma das formas de resolver esta vulnerabilidade está em validar o tamanho da mensagem e o número de caracteres para assim impedir que o programa leia mais caracteres do que o tamanho da mensagem.

## Pergunta P1.3 - Buffer overflow na Heap

O código desenvolvido para a resolução deste exercício encontra-se [aqui](#).

```

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

    if (argc != 2 || strlen(argv[1]) > 10) {
        printf("Número inválido ou tamanho inválido dos argumentos\n");
        return 0;
    }

```

```

    char *dummy = (char *) malloc (sizeof(char) * 10);
    char *readonly = (char *) malloc (sizeof(char) * 10);

```

Neste primeiro excerto de código efetua-se uma validação do *input* do utilizador no sentido em que o número de argumentos passados apenas deverá ser 2, o nome do programa e uma *string*. Relativamente a essa *string* poderia-se optar por 2 escolhas, exigir, com auxílio da função `strlen`, que a mesma tenha comprimento igual ou inferior a 10, de modo a limitar o tamanho do input do utilizador, ou, na alocação da memória, nomeadamente do *array* `dummy` substituir 10 por `strlen(argv[1])`.

```

    strncpy(readonly, "laranjas", strlen("laranjas"));
    strncpy(dummy, argv[1], strlen(argv[1]));

    printf("%s\n", readonly);
}

```

Quanto a este excerto substituiu-se a função de risco `strcpy`, que era vulnerável a *buffer overflows*, pela função `strncpy`, que está protegida contra os mesmos, já que permite controlar o tamanho do que é copiado, sendo esse tamanho definido pela função `strlen`.

## Pergunta P1.4 - Buffer overflow na Stack

O código desenvolvido para a resolução deste exercício encontra-se [aqui](#).

```
int bof(char *str)
{
    char buffer[24];

    if (strlen(str) < strlen(buffer)) {
        strncpy(buffer, str, strlen(str));
    }

    return 1;
}
```

Nesta função simplesmente foi necessário garantir que o tamanho da string que ia ser copiada para o *buffer* não ultrapassa a memória alocada para o mesmo e depois substitui-se a função de cópia, `strcpy` por `strncpy`, já que permite controlar o tamanho do que é copiado, sendo esse tamanho definido pela função `strlen`.

```
int main(int argc, char **argv)
{
    if (argc != 1) {
        printf("Número inválido de argumentos\n");
        return 0;
    }

    char str[517];
    FILE *badfile;

    if((badfile = fopen("badfile", "r")) != NULL){
        fread(str, sizeof(char), 517, badfile);
    }

    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

Esta função, por sua vez, começou-se por limitar o número de argumentos que podem ser passados pelo utilizador, sendo neste caso apenas o programa.

Por último, garantiu-se que a operação `fread` só poderia ser efetuada caso a leitura do ficheiro fosse possível, ou seja, se o ficheiro existir.

# Vulnerabilidade de inteiros

## Pergunta P2.1

1.

A função `vulneravel` apresenta um problema de underflow, pois a variável `tamanho_real` é do tipo `size_t` que corresponde a um tipo de dados sem sinal, ou seja, incapaz de armazenar valores negativos. Se o `tamanho` passado como argumento à função `vulneravel` for igual a 0, então a operação `tamanho_real = tamanho - 1` vai ser um valor negativo o que faz com que o programa não termine corretamente.

2.

Ao colocar o main da seguinte forma, o programa não vai terminar corretamente:

```
1 int main() {  
2     char c[5] = "olaa\0";  
3     vulneravel(c, 0);  
4 }
```

3.

Ao executar o programa com a `main` anterior, o programa dá um erro sendo este "Segmentation fault (core dumped)".

4.

Neste caso é necessário colocar uma validação na condição do `if` que verifique se a variável `tamanho` é superior a 0. Ou seja, aplica-se uma validação de input antes da operação sobre a variável `tamanho_real`:

```
1 void vulneravel (char *origem, size_t tamanho) {  
2     size_t tamanho_real;  
3     char *destino;  
4     if (tamanho < MAX_SIZE && tamanho > 0) {  
5         tamanho_real = tamanho - 1; // Não copiar \0 de origem  
6         ↪ para destino  
7         destino = (char *) malloc(tamanho_real);  
8         memcpy(destino, origem, tamanho_real);  
9     }
```

8 }  
9 }

# Bibliography

- [1] "CWE-787: Out-of-bounds Write" [online]. Disponível em: <https://cwe.mitre.org/data/definitions/787.html> [Acedido em abril de 2022].
- [2] "CVE-2020-0022 Detail" [online]. Disponível em: <https://nvd.nist.gov/vuln/detail/CVE-2020-0022> [Acedido em abril de 2022].