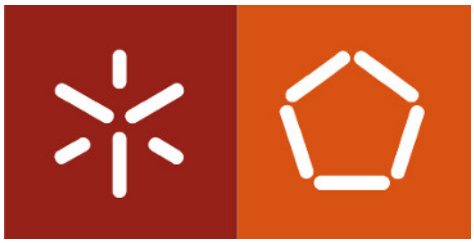

Mestrado em Engenharia Informática
Universidade do Minho

Engenharia de Segurança

Aula 09 TP - 27/04/2020

João Miranda - PG41845
Sandro Cruz - PG41906

27 de Abril de 2020

Conteúdo

Capítulo 1

Buffer Overflow

1.1 Experiência 1.1 - Organização da memória do programa

Utilize o comando Unix size para ver o tamanho, em bytes, do segmento de texto, dados e bss dos programas size1.c, size2.c, size3.c, size4.c e size5.c, seguindo os seguintes passos:

1.1.1 Compile os vários ficheiros. Ex.: gcc -o size1 size1.c

```
user@CSI:~/Desktop/EngSeg/TPraticas/Aula9/codigofonte$ gcc -o size1 size1.c
user@CSI:~/Desktop/EngSeg/TPraticas/Aula9/codigofonte$
```

Figura 1.1: Ficheiro size1.c compilado

1.1.2 Efetue o comando size para cada um dos ficheiros executáveis. Ex.: size size1

```
user@CSI:~/Desktop/EngSeg/TPraticas/Aula9/codigofonte$ size size1
   text    data     bss     dec     hex filename
   1521     544       8    2073     819 size1
user@CSI:~/Desktop/EngSeg/TPraticas/Aula9/codigofonte$
```

Figura 1.2: Verificação do "size"do ficheiro size1

```
user@CSI:~/Desktop/EngSeg/TPraticas/Aula9/codigofonte$ size size2
   text    data     bss     dec     hex filename
   1521     544       8    2073     819 size2
user@CSI:~/Desktop/EngSeg/TPraticas/Aula9/codigofonte$
```

Figura 1.3: Verificação do "size"do ficheiro size2

```
user@CSI:~/Desktop/EngSeg/TPraticas/Aula9/codigofonte$ size size3
   text    data     bss     dec     hex filename
   1521     544      16    2081     821 size3
user@CSI:~/Desktop/EngSeg/TPraticas/Aula9/codigofonte$
```

Figura 1.4: Verificação do "size"do ficheiro size3

```
user@CSI:~/Desktop/EngSeg/TPraticas/Aula9/codigofonte$ size size4
   text    data     bss     dec     hex filename
   1521     548      12    2081     821 size4
user@CSI:~/Desktop/EngSeg/TPraticas/Aula9/codigofonte$
```

Figura 1.5: Verificação do "size"do ficheiro size4

```
user@CSI:~/Desktop/EngSeg/TPraticas/Aula9/codigofonte$ size size5
   text    data     bss     dec     hex filename
   1521     552       8    2081     821 size5
user@CSI:~/Desktop/EngSeg/TPraticas/Aula9/codigofonte$
```

Figura 1.6: Verificação do "size"do ficheiro size5

1.1.3 Verifique e explique as diferenças de valores nos vários segmentos de texto, dados e bss.

As diferenças dos valores dão-se ao facto onde as variareis do código são iniciadas e se é lhes atribuído um valor ou não. Por exemplo o output do "size"nos resultados do ficheiro size1.c e size2.c são iguais dado que a variável iniciada nunca é utilizada nem iniciada no programa e está fora do "scope"função main então o compilador não lhe reserva espaço e os outputs são idênticos, mas o bss deveria ser maior do que o do output do siz1.c. Mas se compararmos o output do ficheiro size1.c e o size3.c os valores de bss e de do hex são diferentes, os valores do bss são superiores porque existem duas variáveis que apenas são inicializadas sem lhes ser atribuído um valor, os valores do output do size2.c e size3.c também seriam diferentes, mas a diferença do valor de bss e hex seriam menores. Agora comparado os valores do output do size3.c com os outputs anteriores, podemos verificar um aumento dos valores de data isto acontece porque uma das variáveis é iniciada, os valores do bss seriam idênticos aos do output do size2.c dado que existe uma variável que não é iniciada. No output do size4.c é idêntico ao do size1.c em termos de bss porque todas as variáveis são iniciadas, mas os valores de data são diferentes dado que tem variáveis iniciadas.

1.2 Experiência 1.2 - Organização da memória do programa

1.2.1 Utilize o programa memoryLayout.c para verificar o layout da memória do programa. Verifique se os resultados são os expectáveis (de notar que as optimizações do compilador e a arquitectura da máquina podem levar a resultados ligeiramente diferentes).

```
user@CSI:~/Desktop/EngSeg/TPraticas/Aula9/codigofonte$ gcc -o memoryLayout memoryLayout.c
user@CSI:~/Desktop/EngSeg/TPraticas/Aula9/codigofonte$ ./memoryLayout
Variáveis de comando linha e ambiente:   0x7fff4647331c 0x7fff46473428 0x7fff46473438
Stack - função main:                     0x7fff4647332c 0x7fff46473328
Stack - função func:                     0x7fff464732dc 0x7fff464732d8 0x7fff464732ec 0x7fff464732e8
Stack - função func2:                    0x7fff464732ac 0x7fff464732a8 0x7fff464732bc 0x7fff464732b8
Heap:                                    0x55bb46c03440
Heap:                                    0x55bb46c03420
Variável global não inicializada:        0x55bb4596f04c 0x55bb4596f050
Variável global inicializada:            0x55bb4596f040 0x55bb4596f044
Text Data:                               0x55bb4576e7bf 0x55bb4576e778 0x55bb4576e740
user@CSI:~/Desktop/EngSeg/TPraticas/Aula9/codigofonte$
```

Figura 1.7: Output da execução do programa memoryLayout.c

Os resultados são os expectáveis dado que os endereços de memoriá das variáveis da "stack"da função main e especialmente das variáveis das funções "func"e "func2"são quase idênticos o que indica que estão próximos no endereçamento da memoria, os resultados das variáveis não iniciadas que estão no bss também tem endereços de memoria parecidos os endereços de memoria das variáveis no bss também são parecidos com os das variáveis iniciadas no ds. As variáveis no heap também partilham endereços parecidos.
Outra factor de notar é o facto de que os endereços de memoria das variáveis do heap estão mais próximas das do bss e ds do que das variáveis da "stack", o que é expectavel dado a forma como a memoria é alocada. Em geral os resultados são exactamente o expectável.

1.3 Experiência 1.3 - Buffer overflow em várias linguagens

Verifique o que ocorre no mesmo programa escrito em Java (LOverflow.java), Python (LOverflow.py) e C++ (LOverflow.cpp), executando-os.

```
user@CSI:~/Desktop/EngSeg/TPraticas/Aula9/codigofonte$ python LOverflow.py
i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
i = 10
Traceback (most recent call last):
  File "LOverflow.py", line 5, in <module>
    buffer[i]=7
IndexError: list assignment index out of range
user@CSI:~/Desktop/EngSeg/TPraticas/Aula9/codigofonte$
```

Figura 1.8: Output do programa LOverflow.py

```
user@CSI:~/Desktop/EngSeg/TPraticas/Aula9/codigofonte$ javac LOverflow.java
user@CSI:~/Desktop/EngSeg/TPraticas/Aula9/codigofonte$ java LOverflow
i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
        at LOverflow.main(LOverflow.java:10)
user@CSI:~/Desktop/EngSeg/TPraticas/Aula9/codigofonte$
```

Figura 1.9: Output do programa LOverflow.java

O LOverflow.cpp também foi testado, mas entrava mantinha-se no loop repetindo os mesmos números.

```
i = 10
i = 7
i = 8
i = 9
i = 10
i = 7
i = 8
i = 9
i = 10
i = 7
i = 8
i = 9
i = 10
i = 7
i = 8
i = 9
i = 10
i = 7
i = 8
```

Figura 1.10: Output do programa LOverflow.cpp

A execução do programa em Java faz o valor de i variar automaticamente entre 0 e 9; em Python varia de 0 a 10; em C++ os valores de i variam ilimitadamente de 7 a 10. Adicionalmente, os programas de Java e de Python terminam com uma exceção porque se tenta aceder a um índice inválido do array. Já com programa em C++ isto não ocorre, pois não existe verificação dos limites do array. Quando o programa escreve na posição 10 do array, que já se encontra fora do limite, está, na realidade, a escrever na variável i, que é alterada para o valor 7, fazendo com que o programa entre assim em loop.

1.4 Pergunta P1.1 - Buffer overflow em várias linguagens

```
user@CSI:~/Desktop/Aula9/codigofonte$ javac LOverflow2.java
user@CSI:~/Desktop/Aula9/codigofonte$ java LOverflow2
Quantos números? 20
Introduza número: 1
Introduza número: 2
Introduza número: 3
Introduza número: 4
Introduza número: 5
Introduza número: 6
Introduza número: 7
Introduza número: 8
Introduza número: 9
Introduza número: 10
Introduza número: 11
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
        at LOverflow2.main(LOverflow2.java:18)
```

Figura 1.11: Output do programa LOverflow2.java

```
user@CSI:~/Desktop/Aula9/codigofonte$ python LOverflow2.py
Quantos numeros? 20
Insira numero: 1
Insira numero: 2
Insira numero: 3
Insira numero: 4
Insira numero: 5
Insira numero: 6
Insira numero: 7
Insira numero: 8
Insira numero: 9
Insira numero: 10
Insira numero: 11
Traceback (most recent call last):
  File "LOverflow2.py", line 5, in <module>
    tests[i]=test
IndexError: list assignment index out of range
```

Figura 1.12: Output do programa LOverflow2.py

```
user@CSI:~/Desktop/Aula9/codigofonte$ g++ -o swap LOverflow2.cpp
user@CSI:~/Desktop/Aula9/codigofonte$ ./swap
Quantos números? 20
Insira número: 1
Insira número: 2
Insira número: 3
Insira número: 4
Insira número: 5
Insira número: 6
Insira número: 7
Insira número: 8
Insira número: 9
Insira número: 10
Insira número: 11
Insira número: 12
Insira número: 13
Insira número: 14
Insira número: 15
Insira número: 16
Insira número: 17
Insira número: 18
Insira número: 19
Segmentation fault
```

Figura 1.13: Output do programa LOverflow2.cpp - Quantidade elevada

```
user@CSI:~/Desktop/Aula9/codigofonte$ ./swap
Quantos números? 10
Insira número: 11
Insira número: 12
Insira número: 13
Insira número: 14
Insira número: 15
Insira número: 16
Insira número: 17
Insira número: 18
Insira número: 19
Insira número: 20
user@CSI:~/Desktop/Aula9/codigofonte$
```

Figura 1.14: Output do programa LOverflow2.cpp - Quantidade menor

No programa Java e no script Python existe um controlo de acesso de memória que impede a escrita numa zona de memória que não esteja previamente declarada para o efeito e, consequentemente, impede vulnerabilidades relacionadas com buffer overflow. Quando se tenta aceder a uma posição de memória que não foi alocada para o array em causa este acesso é negado, sendo apresentada uma

mensagem de erro que indica o acesso efetuado se encontrava fora dos limites. Por outro lado, no programa desenvolvido em C++ o mesmo não se verifica, uma vez que não existe controlo de acesso à memória implícito. Deste modo, quando se tenta aceder a uma posição fora das alocadas para o array o programa não apresenta nenhuma exeção. São possíveis dois comportamentos distintos, dependendo do input introduzido onde o programa é forçado a parar, devolvendo Segmentation Fault caso se coloque uma quantidade elevada de números, ou termina o programa com um teste de quantidade de 10 elementos p.e.

1.5 Experiência 1.4 - Buffer overflow em várias linguagens

```
user@CSI:~/Desktop/Aula9/codigofonte$ javac LOverflow3.java
user@CSI:~/Desktop/Aula9/codigofonte$ java LOverflow3
Quantos valores quer guardar no array?
20
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
        at LOverflow3.main(LOverflow3.java:15)
```

Figura 1.15: Output do programa LOverflow3.java - Tamanho a 20.

```
user@CSI:~/Desktop/Aula9/codigofonte$ java LOverflow3
Quantos valores quer guardar no array?
10
Que valor deseja recuperar?
3
O valor é 7
```

Figura 1.16: Output do programa LOverflow3.java - Tamanho a 10.

```
user@CSI:~/Desktop/Aula9/codigofonte$ python LOverflow3.py
Quantos valores quer guardar no array? 20
Traceback (most recent call last):
  File "LOverflow3.py", line 4, in <module>
    vals[i] = count-i
IndexError: list assignment index out of range
```

Figura 1.17: Output do programa LOverflow3.py - Tamanho a 20.

```
user@CSI:~/Desktop/Aula9/codigofonte$ python LOverflow3.py
Quantos valores quer guardar no array? 10
Que valor deseja recuperar? 2
O valor e  8
```

Figura 1.18: Output do programa LOverflow3.py - Tamanho a 10.

```
user@CSI:~/Desktop/Aula9/codigofonte$ g++ -o swap LOverflow3.cpp
user@CSI:~/Desktop/Aula9/codigofonte$ ./swap
Quantos valores quer guardar no array? 20
Que valor deseja recuperar? 15
O valor é 0
```

Figura 1.19: Output do programa LOverflow3.cpp - Quantidade pequena

```
user@CSI:~/Desktop/Aula9/codigofonte$ ./swap
Quantos valores quer guardar no array? 100000
Segmentation fault
```

Figura 1.20: Output do programa LOverflow3.cpp - Quantidade elevada

O que sucede nesta situação é semelhante ao que acontecia anteriormente, no que diz respeito ao controlo de acesso à memória, onde nos programas desenvolvidos em Java e Python o acesso a regiões de memória fora do que foi declarado são impedidos e no no programa C++ é permitido o acesso a posições de memória fora das alocadas. A execução dos programas LOverflow3.java e LOverflow3.py resulta em execções semelhantes às anteriormente apresentadas. No caso do programam desenvolvido em Java, é apresentado um erro quando é requisitado um valor presente no índice superior ao 10, dado que apenas são alocadas dez posições para o array.

1.6 Experiência 1.5 - Buffer overflow em várias linguagens

```
user@CSI:~/Desktop/Aula9/codigofonte$ cat /tmp/temps.txt
30.0 30.1 30.2 30.3 30.4 30.5 30.6 20.7 30.8 30.9
user@CSI:~/Desktop/Aula9/codigofonte$ javac ReadTemps.java
user@CSI:~/Desktop/Aula9/codigofonte$ java ReadTemps
Foram lidas 10 temperaturas.
user@CSI:~/Desktop/Aula9/codigofonte$ python ReadTemps.py
Foram lidas 10 temperaturas.
```

Figura 1.21: Output do programa ReadTemp.java e ReadTemp.py - Teste 1.

```
user@CSI:~/Desktop/Aula9/codigofonte$ cat /tmp/temps.txt
30.0 30.1 30.2 30.3 30.4 30.5 30.6 20.7 30.8 30.9 31.0 31.2
user@CSI:~/Desktop/Aula9/codigofonte$ java ReadTemps
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
        at ReadTemps.main(ReadTemps.java:15)
user@CSI:~/Desktop/Aula9/codigofonte$ python ReadTemps.py
Traceback (most recent call last):
  File "ReadTemps.py", line 6, in <module>
    temps[numTemps] = float(i)
IndexError: list assignment index out of range
```

Figura 1.22: Output do programa ReadTemp.java e ReadTemp.py - Teste 2.

Relativamente às temperaturas 30.0 30.1 30.2 30.3 30.4 30.5 30.6 20.7 30.8 30.9, tanto os programas em Java como em Python conseguem ler as 10 temperaturas. No entanto, ultrapassando esse valor, tal já não se verifica e ambos retornam erro.

1.7 Pergunta P1.2 - Buffer overflow

1.7.1 RootExploit.c

```
user@CSI:~/Desktop/Aula9/codigofonte$ gcc -o RootExploit RootExploit.c
RootExploit.c: In function ‘main’:
RootExploit.c:10:5: warning: implicit declaration of function ‘gets’ [-Wimplicit-function-declaration]
     gets(buff);
     ^~~~
/tmp/cc8qkcRo.o: In function `main':
RootExploit.c:(.text+0x28): warning: the `gets' function is dangerous and should not be used.
```

Figura 1.23: Compilação do programa RootExploit.c

```
user@CSI:~/Desktop/Aula9/codigofonte$ ./RootExploit

 Insira a password de root:
password

 Password errada

 Foram-lhe atribuidas permissões de root/admin
```

Figura 1.24: Output do programa RootExploit.c

Neste programa a vulnerabilidade de buffer overflow existe porque a função gets não valida o tamanho do input, e provém de se de se introduzir uma password sem o limite mínimo de 5 caracteres exigidos (char buff[4]). Desta forma, é possível escrever na variável pass caso se insira um input com tamanho superior a 4. Assim, para obter a mensagem "São atribuidas permissões de admin"basta que a variável pass tenha uma valor diferente de 0, ou seja, basta apenas inserir na password uma string com 5 caracteres, mesmo que a password esteja errada.

1.7.2 0-simple.c

```
user@CSI:~/Desktop/Aula9/codigofonte$ gcc -o 0-simple 0-simple.c
0-simple.c: In function ‘main’:
0-simple.c:16:3: warning: implicit declaration of function ‘gets’ [-Wimplicit-function-declaration]
   gets(buffer);
   ^~~~
/tmp/cczmhmRK.o: In function `main':
0-simple.c:(.text+0x2f): warning: the `gets' function is dangerous and should not be used.
```

Figura 1.25: Compilação do programa 0-simple.c

```
user@CSI:~/Desktop/Aula9/codigofonte$ ./0-simple
You win this game if you can change variable control'
password
Try again...
```

Figura 1.26: Output do programa 0-simple.c - Perder o jogo.

```
user@CSI:~/Desktop/Aula9/codigofonte$ ./0-simple
You win this game if you can change variable control'
214214124wfwfwafwafawfwfafwafafwfwafwfwafawfeegergeryry2464262464262tfqetfqertqrqrqe1413543153fqfqefqfeqq
YOU WIN!!!
Segmentation fault
```

Figura 1.27: Output do programa 0-simple.c - Ganhar o jogo.

No ficheiro 0-simple.c a vulnerabilidade existe e é explorada pelas mesmas razões do exemplo anterior, no entanto, o tamanho do input necessário para explorar a vulnerabilidade tem que ser superior em mais do que uma unidade ao tamanho do buffer. Este facto pode estar relacionado com o alinhamento de memória forçado pelo compilador. A vulnerabilidade neste programa advém assim de não se ter em consideração se o input é maior que o buffer (char buffer[64]),tal como é exigido. Facilmente alterando a variável control para um valor diferente de 0, i.e., introduzindo caracteres que ultrapassem esse limite, é simples obter a mensagem "YOU WIN!!!".

1.8 Experiência 1.6 - Formatted I/O

```
user@CSI:~/Desktop/Aula9/codigofonte$ gcc -o IOformatado IOformatado.c
user@CSI:~/Desktop/Aula9/codigofonte$ ./IOformatado
Leitura segura: %d
ECO |%d
Leitura vulneravel: %d
-844130448
user@CSI:~/Desktop/Aula9/codigofonte$ ./IOformatado
Leitura segura: %s
ECO |%s
Leitura vulneravel: %s

user@CSI:~/Desktop/Aula9/codigofonte$ ./IOformatado
Leitura segura: %p
ECO |%p
Leitura vulneravel: %p
0x7ff74d3ef770
```

Figura 1.28: Output do programa IOformatado.c

Na função segura(), o output da função printf é formatado pela string "%s", onde qualquer sequência de formatação no input ("%s", "%d", "%p",...) será lida de forma literal, i.e. o output não será expandido para substituir as ocorrências destes padrões. Na função vulnerável(), como a string passada à função printf é exatamente aquela que é lida, caso esta contenha alguma sequência de formatação, esta vai ser substituída por um valor indeterminado, uma vez que não é passado nenhum argumento adicional.

1.9 Pergunta P1.3 - Read overflow

```
user@CSI:~/Desktop/Aula9/codigofonte$ gcc -o ReadOverflow ReadOverflow.c
user@CSI:~/Desktop/Aula9/codigofonte$ ./ReadOverflow
Insira numero de caracteres: 5
Insira frase: 12345
ECO: |12345|
Insira numero de caracteres: 5
Insira frase: 123
ECO: |123..|
Insira numero de caracteres: 5
Insira frase: 123456
ECO: |12345|
```

Figura 1.29: Output do programa ReadOverFlow.c

O programa ReadOverflow.c é responsável por fazer echo de um determinado número de caracteres previamente definidos pelo utilizador. A primeira característica é o facto de que o valor de p passa a ser igual ao valor de buff, caso a função fgets não retorne NULL. Outra característica é a de que o programa não verifica qual o tamanho máximo permitido de carateres a serem introduzidos como input. Assim, a vulnerabilidade existente neste programa deve-se ao facto de este ler mais informação que a que era suposta. Se o utilizador só escolher inserir 6 caracteres, por exemplo, não deveria ser possível inserir mais, pois o programa iria ler bytes de memória fora da stack (pois o buffer é a primeira variável local do programa). Isso seria particularmente perigoso, pois o utilizador ao inserir um valor maior que o devido poderia adquirir informação de zonas de memória não permitidas, com conteúdo sensível.

1.10 Experiência 1.7 - Buffer overflow na Heap

```
user@CSI:~/Desktop/EngSeg/TPraticas/Aula9/codigofonte$ ./overflowHeap
Segmentation fault
user@CSI:~/Desktop/EngSeg/TPraticas/Aula9/codigofonte$
```

Figura 1.30: Output do programa overflowHeap.1.c

```
user@CSI:~/Desktop/EngSeg/TPraticas/Aula9/codigofonte$ ./overflowHeap2
Endereço da variavel dummy: 0x55eb35242010
Endereço da variavel readonly: 0x55eb35242030
Segmentation fault
```

Figura 1.31: Output do programa overflowHeap.2.c

```
user@CSI:~/Desktop/EngSeg/TPraticas/Aula9/codigofonte$ ./overflowHeap2 1
Endereço da variavel dummy: 0x5619377ed010
Endereço da variavel readonly: 0x5619377ed030
laranjas
```

Figura 1.32: Output do programa overflowHeap.2.c, mas recebendo parâmetros

O resultado do output do programa overflowHeap.1.c recebendo parâmetros é idêntico.
Os problemas existentes de overflow neste caso existem porque o variável "dummy"tenta fazer "stringcopy"de uma variável um valor null, o que faz com que exista um erro de segmentação, isto pode ser evitando criando um valor por defeito caso não haja parâmetros passados para a função.

```
user@CSI:~/Desktop/EngSeg/TPraticas/Aula9/codigofonte$ cat overflowHeap.2.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    char *dummy = (char *) malloc (sizeof(char) * 10);
    char *readonly = (char *) malloc (sizeof(char) * 10);

    printf("Endereço da variavel dummy: %p\n", dummy);
    printf("Endereço da variavel readonly: %p\n", readonly);

    printf("argv[1]: %s\n", argv[1]);

    strcpy(readonly, "laranjas");
    if(!argv[1]){
       strcpy(dummy, "laranja");
    }else{
       strcpy(dummy, argv[1]);
    };
    printf("%s\n", readonly);
}
user@CSI:~/Desktop/EngSeg/TPraticas/Aula9/codigofonte$
```

Figura 1.33: Alterações propostas para impedir o buffer overflow no heap

```
user@CSI:~/Desktop/EngSeg/TPraticas/Aula9/codigofonte$ gcc -o overflowHeap2 over
flowHeap.2.c
user@CSI:~/Desktop/EngSeg/TPraticas/Aula9/codigofonte$ ./overflowHeap2
Endereço da variavel dummy: 0x55cd07d93010
Endereço da variavel readonly: 0x55cd07d93030
argv[1]: (null)
laranjas
user@CSI:~/Desktop/EngSeg/TPraticas/Aula9/codigofonte$ ./overflowHeap2 1
Endereço da variavel dummy: 0x55631c76e010
Endereço da variavel readonly: 0x55631c76e030
argv[1]: 1
laranjas
user@CSI:~/Desktop/EngSeg/TPraticas/Aula9/codigofonte$
```

Figura 1.34: Output das alterações feitas

1.11 Experiência 1.8 - Buffer overflow na Stack

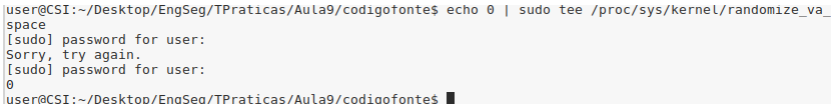

Figura 1.35: Desactivar o ASL

Figura 1.36: Compilar overflowStack.1.c com informação de debug, a ser utilizada pelo GDB

Figura 1.37: Iniciar o debug do programa com o gdb

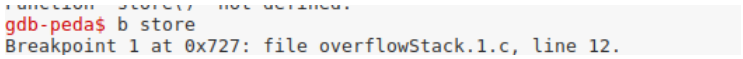

Figura 1.38: No gdb, colocar breakpoint na função store()

Figura 1.39: No gdb, executar o programa até ao breakpoint

```
gdb-peda$ info f
Stack level 0, frame at 0x7fffffffe0a0:
 rip = 0x555555554727 in store (overflowStack.1.c:12); saved rip = 0x55555555475f
 called by frame at 0x7fffffffe0c0
 source language c.
 Arglist at 0x7fffffffe090, args: valor=0x0
 Locals at 0x7fffffffe090, Previous frame's sp is 0x7fffffffe0a0
 Saved registers:
  rbp at 0x7fffffffe090, rip at 0x7fffffffe098
gdb-peda$
```

Figura 1.40: O endereço de memória onde está guardada a função debug()

```
gdb-peda$ p debug
$1 = {void ()} 0x5555555546f0 <debug>
```

Figura 1.41: O endereço de memória onde está guardada a função debug()

```
gdb-peda$ p &buf
$2 = (char (*)[10]) 0x7fffffffe086
```

Figura 1.42: No array buf para reescrever o %rip, veja em que endereço se inicia o array buf (eia)

1.12 Pergunta P1.4

1.12.1 Agora que já tem experiência em efetuar o overflow a um buffer (cf. pergunta P1.3), consegue fazer o mesmo se for necessário um valor exato?

Primeiro começamos por descobrir qual é o valor em descodificado do hexadecimal "0x61626364"que ficamos a descobrir que é o correspondente a "abdc", então demos overflow só com uma letra até descobrirmos quando começava a dar overflow para a variável que pretendíamos, aí tivemos de descobrir se as

escrita na memória é feita seguindo a *little-endian* ou *big-endian*, descobrindo que é feita em *little-endian* demos ao programa o *input* seguinte:
"xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxdcba"
que nos deu o output de "Congratulations, you win!!!".

1.13 Experiência 1.9

Testamos de diferentes forma invocar a função "win", mas tivemos pouco sucesso. Utilizamos ferramentas do género do "objdump"e o "gdb"mas não conseguimos com os dados recolhidos chegar a um ponto em que conseguimos invocar a função. Verificamos com essas ferramentas a localização em memoria da função e até inserimos inputs nós achamos que com essas informações deveríamos ser capazes de invocar, mas depararmo-nos constantemente com um erro de segmentação que nós impossibilitava a execução da função.
Achamos que é possível, apenas não conseguimos.

1.14 Pergunta P1.5 - Buffer overflow na Heap

```
user@CSI:~/Desktop/EngSeg/TPraticas/Aula9/codigofonte$ cat overflowHeap.1.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

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

    if(argc == 1) {
      errx(1, "please specify an argument\n");
    };
    if(sizeof(argv[1]) <= 1000000){
       errx(1, "Input a smaller input\n");
    };

    char *dummy = (char *) malloc (sizeof(char) * sizeof(argv[1]));
    char *readonly = (char *) malloc (sizeof(char) * 10);

    strcpy(readonly, "laranjas");
    strcpy(dummy, argv[1]);
    printf("%s\n", readonly);
}
```

Figura 1.43: Alterações feitas ao overflowHeader.1.c para ser mais seguro.

A primeira alteração que fizemos foi exigir um input do utilizador de forma a que não cause o overflow do heap, caso o utilizador não insira um input o programa fecha, depois é verificado o tamanho do input, isto não necessariamente necessário mas achamos que seria melhor por um limite máximo do tamanho do input, se ultrapassarem o limite o programa fecha.
No "malloc"da variável "readonly"utilizamos o tamanho dos argumentos de forma a que não acontecem erros no "heap".

1.15 Pergunta P1.6 - Buffer overflow na Stack

```
user@CSI:~/Desktop/EngSeg/TPraticas/Aula9/codigofonte$ cat code.c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str)
{
        char buffer[sizeof(str)];
        strcpy(buffer, str);
        return 1;
}

int main(int argc, char **argv)
{
        char str[517];
        FILE *badfile;
        badfile = fopen("badfile", "r");
        if(sizeof(badfile) > 517){
                errx(1, "The file that you select is to large\n");
        };
        fread(str, sizeof(char), 517, badfile);
        bof(str);
        printf("Returned Properly\n");
        return 1;
}
user@CSI:~/Desktop/EngSeg/TPraticas/Aula9/codigofonte$
```

Figura 1.44: Alterações feitas ao code.c para ser mais seguro.

Primeiro implementamos um "if"que verfica o tamanho do ficheiro que é recebido, se ele for maior do que tamanho da string "str", de forma a não haver um overflow, também no "buffer"da função "bof"o tamanho desse buffer é definido pelo tamanho recebido pelo parâmetro "str", assim não existe riscos de existir um overflow do array.