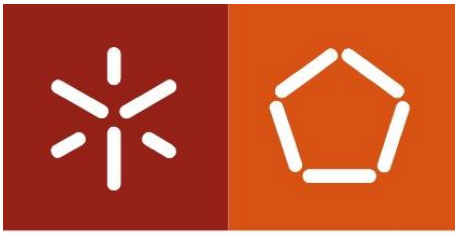

Universidade do Minho

Departamento de Informática

Mestrado em Engenharia Informática

Criptografia e Segurança de Informação

Engenharia de Segurança
Ficha Prática 11

Grupo Nº 3

Ariana Lousada (PG47034) Luís Carneiro (PG46541)
Rui Cardoso (PG42849)

31 de maio de 2022

1 Buffer Overflow

1.1 Pergunta 1.1 - Buffer Overflow

Analise e teste os programs escritos em C RootExploit.c e 0-simple.c. Indique qual a vulnerabilidade de Buffer Overflow existente e o que tem de fazer (e porquê) para a explorar e (i) obter a confirmação de que lhe foram atribuídas permissões de root/admin, sem utilizar a password correta, (ii) obter a mensagem "YOU WIN!!!".

A vulnerabilidade de ambos os programas de Buffer Overflow deve-se ao facto de ambos utilizarem a função 'gets()' para receber o valor que é escrito no buffer. Esta função não verifica se a variável 'buffer' tem memória alocada suficiente para receber o valor especifico, caso um atacante envie como argumento uma string de tamanho maior ao do buffer (neste caso 64 para o programa 0-simple.c e 4 para o RootExploit.c), vai acontecer buffer overflow o que permite modificar partes da memória que não deveriam ser acedidos.

Overflow no programa 0-simple.c

```
dev@dev-ubuntu:~/Desktop/EngSeg-main/EngSeg-main/Pratica1/Aula12.a/codigofonte$
./0-simple
You win this game if you can change variable control'
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIILLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSSTTTTUUUUV
YOU WIN!!!
```

Overflow no programa RootExploit.c

```
dev@dev-ubuntu:~/Desktop/EngSeg-main/EngSeg-main/Pratica1/Aula12.a/codigofonte$
./RootExploit
Endereço da variavel pass: 0x7ffc10f88cbc
Endereço da variavel buff: 0x7ffc10f88cb8

 Insira a password de root:
ABCDE

 Password errada
Valor de pass: 69

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

Nota: Os programas foram compilados com a flag '-fno-stack-protector' caso contrário o compilador cria proteção contra overflow.

1.2 Pergunta 1.2 - Read Overflow

Analise e teste o programa escrito em C ReadOverflow.c. O que pode concluir?

O programa ReadOverflow.c tem proteção contra overflow pois utiliza a função fgets() em vez de gets(). Deste modo o programa apenas lê o tamanho máximo que cabe no buffer.

```
dev@dev-ubuntu:~/Desktop/EngSeg-main/EngSeg-main/Pratica1/Aula12.a/codigofonte$
./ReadOverflow
Insira numero de caracteres: 5
Insira frase: 123456789
ECO: |12345|
```

1.3 Pergunta 1.3 - Buffer overflow na Heap

Utilize as várias técnicas de programação defensiva introduzidas na aula teórica para mitigar as vulnerabilidades de Buffer overflow na heap visto na aula teórica (ficheiro overflowHeap.1.c). Explique as alterações que fez.

Na seguinte imagem pode-se ver o código com as alterações.

```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);

    strcpy(readonly, "laranjas");
    if(argc > 1){
        if(strlen(argv[1]) > 10){
            printf("Na versao antiga causaria overflow...\n");
        }
        strncpy(dummy, argv[1], 5); // só copia se argv[1] existir
    }

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

O valor recebido como argumento é copiado para a variável `dummy` através da função strncpy(), que toma em consideração o tamanho da variável e impede overflow.

Exemplo:

```
dev@dev-ubuntu:~/Desktop/EngSeg-main/EngSeg-main/Pratica1/Aula12.a/codigofonte$
./overflowHeap.1 123456789010
Na versao antiga causaria overflow...
laranjas
```

1.4 Pergunta 1.4 - Buffer overflow na Stack

Em https://github.com/npapernot/buffer-overflow-attack encontra o programa stack.c com um problema de buffer overflow. Utilize as várias técnicas de programação defensiva introduzidas na aula teórica para mitigar essas vulnerabilidade Explique as alterações que fez.

Na seguinte imagem pode-se ver o código com as alterações.

```c
C stack.c > ...
4   #include <stdlib.h>
5   #include <stdio.h>
6   #include <string.h>
7   int bof(char *str)
8   {
9       char buffer[24];
10      /* The following statement no longer has a buffer overflow problem */
11
12      strncpy(buffer, str, 24);
13      return 1;
14  }
15
16  int main(int argc, char **argv)
17  {
18      char str[517];
19      FILE *badfile;
20      badfile = fopen("badfile", "r");
21      fread(str, sizeof(char), 517, badfile);
22      bof(str);
23      printf("Returned Properly\n");
24      return 1;
25  }
```

Na linha 12 o método de copiar a str para o buffer foi alterado sendo este primeiro(strcpy(buffer, str))o que tinha vulnerabilidade de overflow para (strncpy(buffer, str, 24)), que toma em consideração o tamanho da variável buffer.

```
dev@dev-ubuntu:~/Desktop/buffer-overflow-attack-master$ ./stack
Returned Properly
```

2 Vulnerabilidade de inteiros

2.1 Pergunta 2.1

Analise o programa underflow.c.

2.1.1 Qual a vulnerabilidade que existe na função vulneravel() e quais os efeitos da mesma?

A função `vulneravel()` possui uma vulnerabilidade de *underflow*. Esta vulnerabilidade pode resultar de uma operação aritmética(no caso deste ficheiro, na linha 12: `tamanho_real = tamanho - 1;`) que resulta num valor que excede os limites máximos suportados por um determinado tipo de dados(no caso deste ficheiro, do `size_t`).

Neste caso em particular pode levar à escrita na memória o número errado de bits com a função `memcpy`, assim como no local errado, visto que a variável `tamanho_real` é também utilizada para definir o `destino`, que por sua vez define o local de escrita na memória.

2.1.2 Complete o main() de modo a demonstrar essa vulnerabilidade.

Após uma breve pesquisa, a equipa de trabalho notou que o limite máximo do *size_t* corresponde à seguinte macro: `__SIZE_MAX__`.

Ao utilizar este valor para definir o `tamanho` a ser utilizado na função obtém-se o seguinte:

```
1  int main() {
2      size_t tamanho = -__SIZE_MAX__-1;
3      vulneravel("helloThere", tamanho);
4  }
```

2.1.3 Ao executar dá algum erro? Qual?

Ao executar o programa, este dá um erro de *segmentation fault*:

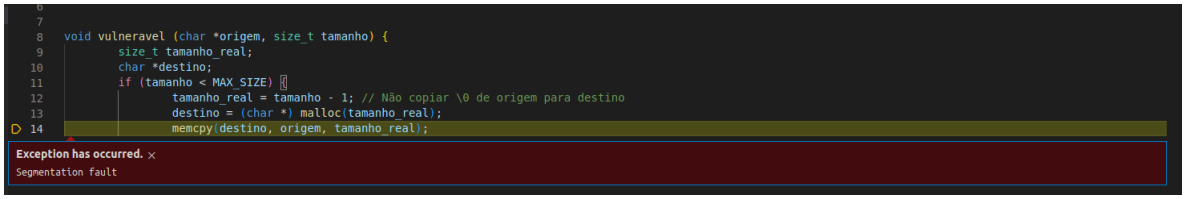

Figura 1: *Segmentation Fault*

Este erro resulta da tentativa de escrita de um número inválido de bits em memória, assim como na incorreta definição da variável `destino` que resulta da incorreta definição da variável `tamanho`, que por sua vez indica o local de escrita na memória.

2.1.4 Utilize as várias técnicas de programação defensiva introduzidas na aula teórica para mitigar as vulnerabilidades. Explique as alterações que fez.

De modo a evitar casos de *underflow* e *overflow*, a técnica a aplicar mais direta será a validação de valores de input da função.

Com isto em conta, adicionou-se o seguinte excerto de código à função `vulneravel()`. [1]

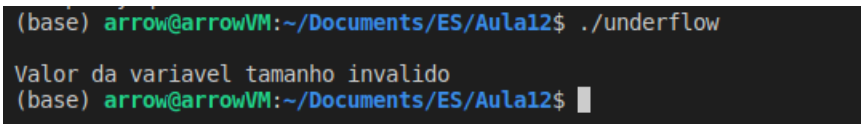

Figura 2: Execução do mesmo exemplo da figura 1 com alterações no ficheiro `underflow.c`

[1] O ficheiro com as alterações aplicadas pode ser consultado na diretoria *AP1/Ficha-11/Pergunta2-1* no repositório GitHub do grupo de trabalho.