Universidade do Minho

Mestrado Integrado em Engenharia Informática

Engenharia de Segurança

Trabalho TP7

Grupo 2
Paulo Gameiro - A72067
Pedro Rodrigues - PG41092
Rafaela Soares - A79034

Braga, Portugal
27 de Abril de 2020

Conteúdo

1 Pergunta P1.1 - Buffer overflow em várias linguagens

Após a análise dos programas, evidenciou-se que estes têm a mesma rotina, logo o que seria de esperar é que os resultados fossem os mesmos. No entanto, quando se insere um número maior do que o esperado pelo programa, todos eles apresentam reações diferentes.

1.1 Programa escrito em C++ (LOverflow3.cpp)

Quando se corre o programa em C++ e fazemos *Overflow* da variável *tests*, o mesmo irá continuar a pedir para inserir o próximo número, mas não deixará colocar qualquer *input* até chegar ao fim da quantidade de números que tenhamos dado. No entanto, quando fazemos com que o mesmo ocorra, quando nos pede para inserirmos a quantidade de números que queremos colocar, o programa entra em *loop* a fazer *print* constante de "Insira numero: ".

1.2 Programa escrito em Java (LOverflow3.java)

Ao correr o programa em Java e tentamos fazer o mesmo, ocorre uma *InputMismatchException* e o programa é terminado.

1.3 Programa escrito em Python (LOverflow3.py)

Quando se corre o programa em Python e tentamos fazer *Overflow*, é obtido o erro *OverflowError: range() result has too many items*, mas somente quando o fazemos no input de "Quantos numeros?", visto que "Insira numero: "aceita como *input* números muito maiores.

2 Pergunta P1.2 - Buffer overflow

Ao executarmos o primeiro programa *RootExploit.c*, se tentarmos fazer o ataquede *BufferOverflow* no *input* pedido, este irá terminar a execução fazendo *printf* das seguintes mensagens:

- *password errada*
- *Foram-lhe atribuídas permissões de root*

Ao replicar o mesmo no programa *0-simple.c* obtemos a mensagem *YOU WIN !!!*.É de notar que o tamanho do input dado para os dois programas é diferente pois no primeiro o tamanho do *buffer* é 4 caracteres enquanto que no segundo é de 64.

3 Pergunta P1.3 - Read overflow

Inicialmente, o programa **ReadOverflow.c** pede ao utilizador que indique qual o número de caracteres que vai inserir, sendo que, quando é recebido o *input* do utilizador, não é feita qualquer verificação sobre se o número de caracteres a introduzir é maior que o tamanho máximo da variável **buf** (100).

Figura 1: Resultado do Read Overflow

A introdução na variável **p** de um valor superior ao tamanho alocado na variável **buf**, significa que o ciclo vai percorrer o **buf** e imprimir mais caracteres do que aqueles que pertencem à variável, permitindo, assim, ler dados que se encontram nos endereços de memória seguintes, já fora da variável.

4 Pergunta P1.4

Durante a execução do programa **1-match**, recebemos a informação de que é necessário alterar o valor da variável de controlo para **0x61626364** para que consigamos ganhar o jogo e, como tal, obter o **"Congratulations"** que se pretende.

```
root@CSI:~/EngSeg/TPraticas/Aula9/codigofonte# ./1-match teste
You win this game if you can change variable control to the value 0x61626364'
Try again, you got 0x00000000
```

Figura 2: Execução do programa com parâmetro incorreto

Tendo em conta que estamos perante um sistem UNIX, a arquitetura é **little-endian**, o que significa que os *bytes* são guardados em endereços sucessivos da memória por ordem crescente do seu valor.

Partindo do hexadecimal **61626364** que nos é mostrado, é possível perceber que o valor corresponde a **abcd** e que o buffer tem 64 *bytes*, ou seja, através da soma do tamanho do buffer mais o seu endereço obtém-se o valor de 72 *bytes*.

Aos 72 *bytes*, é necessário ainda ter em conta o tamanho do endereço de retorno ($4bytes$), o que nos dá o tamanho final de 76 *bytes*.

Ou seja, de forma a alterar a variável de controlo para o valor **0x61626364**, é necessário passar 76 caracteres mais a string "abcd", que como estamos perante uma arquitetura **little-endian** tem de ser passada com **dcba**.

```
root@CSI:~/EngSeg/TPraticas/Aula9/codigofonte# ./1-match 12345678912345678912345678900012345678901234567890123456789012345678901234567890123456dcba
You win this game if you can change variable control to the value 0x61626364'
Congratulations, you win!!! You correctly got the variable to the right value
root@CSI:~/EngSeg/TPraticas/Aula9/codigofonte#
```

Figura 3: Execução do programa com parâmetro correto

5 Pergunta P1.5

No intuito de se mitigar as vulnerabilidades de *Buffer overflow* na *heap* evidenciadas na aula teórica correspondente ao presente trabalho prático, utilizou-se como técnicas de programação defensiva a alocação de memória e espaço alocado.

5.1 Alocação de memória

No que diz respeito à programação defensiva alocação de memória, advoga-se que se deve alocar memória, sempre que possível, depois de se saber a quantidade de memória que se necessita.

Como o apontador dummy irá guardar o que se encontra em argv[1], consegue-se obter facilmente o tamanho de argv[1] e, consequentemente, saber qual é a quantidade de memória que se necessita para o guardar. Desta forma, consegue-se garantir que o tamanho de argv[1] não é superior ao que a variável dummy pode guardar, pelo que não irá subscrever na variável readonly, o que não acontece na implementação do ficheiro overflowHeap.1.c.

Nesse sentido, em primeiro lugar, declarou-se uma variável argv_size, de tipo int, que guarda o tamanho de argv[1]. Posteriormente, alterou-se na variável dummy o valor de 10 para argv_size, de forma a garantir que se está a alocar o espaço necessário para guardar argv[1] na mesma.

Da mesma forma que se tem conhecimento que se quer guardar na variável dummy o que se encontra em argv[1], também se sabe que se quer guardar na variável readonly a *string* "laranjas".

Posto isto, facilmente se consegue inferir qual é o tamanho da *string* laranjas e modificar o tamanho que foi previamente dado à variável readonly, de forma a garantir que esta consegue armazenar a *string* "laranjas", aplicando a técnica de programação defensiva alocação de memória.

Deste modo, declarou-se uma variável laranjas_size, de tipo int, que guarda o tamanho *string* "laranjas". Posteriormente, alterou-se na variável readonly o valor de 10 para laranjas_size.

5.2 Espaço alocado

Decidiu-se também aplicar a técnica de programação defensiva espaço alocado, em que, antes de se copiar os dados, garante-se que a variável de destino tem espaço suficiente para guardar os dados.

Para isso, verifica-se se o tamanho da *string* laranjas é superior ao que a variável readonly pode guardar(laranjas_size). Caso seja, não é possível guardar na variável readonly. Caso não seja, é possível. Tal se sucede de forma semelhante relativamente ao armazenamento de argv[1] em dummy.

Apesar da técnica alocação de memória garantir que as variáveis têm espaço suficiente para aquilo que se pretende guardar e, consequentemente, "impedirem" que uma variável subscreva numa variável de endereço de memória posterior ao seu, considerou-se pertinente aplicar a técnica espaço alocado na mesma, não só a título instrutivo para o grupo, mas também para consolidar a programação defensiva.

Nota: De referir que se ponderou não utilizar a função strcpy, uma vez que esta não é segura, pelo facto de não ter um comportamento definido para casos em que o tamanho do destino não seja o suficiente para guardar o que se pretende. Uma alternativa seria aplicar a técnica de programação defensiva referente a não se utilizar funções de risco, sendo como uma alternativa a função strlcpy. Tal não foi testado nem implementado porque não se conseguiu instalar a biblioteca libbsd.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>


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

    int argv_size = strlen(argv[1]);

    int laranjas_size = strlen("laranjas");


    char *dummy = (char *) malloc (sizeof(char) * argv_size);
    char *readonly = (char *) malloc (sizeof(char) * laranjas_size);


    if ((strlen("laranjas")) > laranjas_size) {
        printf("Not enough space!");
        return -1;
    }

    else {
        strcpy(readonly,"laranjas");
    }

    if ((strlen(argv[1])) > argv_size) {
        printf("Not enough space for argv[1]!");
        return -1;
    }
    strcpy(dummy, argv[1]);
    printf("%s\n", readonly);

}
```

6 Pergunta P1.6

No âmbito de se mitigar as vulnerabilidades de *Buffer overflow* na *stack* que se encontram em `https://github.com/npapernot/buffer-overflow-attack`, precisamente no programa stack.c, utilizou-se como técnica de programação defensiva espaço alocado.

Uma vez que não existe qualquer tipo de verificação acerca do que a função fopen retorna, é necessário fazê-lo, uma vez que pode ocorrer *segmentation fault*, caso o ficheiro não exista e se tente executar a leitura no mesmo.

Outra ocorrência de *buffer overflow* que pode surgir deve-se ao facto de não ser verificado se o buffer tem espaço o suficiente para armazenar a variável str. Como a variável str pode ter até 517 *bytes* e o buffer só consegue armazenar 24 *bytes*, existe a possibilidade de ocorrer *buffer overflow*, caso o tamanho da variável str seja superior a 24 *bytes*. Nesse sentido, é necessário verificar se o tamanho da variável str é superior a 24. Caso o seja, não é possível guardar no buffer. É, assim, aplicado a técnica de programação defensiva de espaço alocado - antes de se copiar os dados, garante-se que a variável de destino tem espaço suficiente para guardar os dados.

De notar que também se ponderou não utilizar a função strcpy. O motivo pelo qual não se o fez já foi explicado na pergunta anterior.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>




int bof(char *str) {

    char buffer[24];

    if((strlen(str)) > 24) {
        printf("Not enough space\n");
        return -1;
    }

    strcpy(buffer, str);

    return 1;
}
```

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

    char str[517];
    FILE *badfile;

    badfile = fopen("badfile.txt", "r");


    if(badfile == NULL) {
         perror("open");
         return -1;
    }

    fread(str, sizeof(char), 517, badfile);

    bof(str);

    printf("Returned Properly\n");

    return 1;

}
```