

Sécurité applicative

Laboratoire 5

Sécurité Applicative - Prévention

Adrien Voisin, Bastien Bodart

IR313 - Henallux 2021-2022

1 Memory Leak

Dans certains langages comme le C, la bonne gestion de la mémoire est laissée à l'initiative du développeur. Des erreurs de programmation peuvent mener à de graves problèmes sur un système. L'exemple le plus courant est la fuite mémoire où, un programme tournant en tâche de fond va petit à petit consommer la mémoire d'un serveur jusqu'à le rendre inutilisable.

1.1 Mise en situation (exercice à réaliser en groupe)

Vous êtes contactés par votre équipe système qui se plaint de devoir redémarrer tous les jours un serveur car sa mémoire est saturée après une journée de fonctionnement. Elle pense avoir identifié la cause et vous envoie le code source d'un programme (*sorter.c*) utilisé pour ordonnancer des tâches sur le serveur.

- Analysez de manière dynamique la fuite mémoire à l'aide d'outils comme Valgrind^[1] et top^[2]

Le code :

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

void swap(int array[], int n,int i, int y)
{
    int *temp_array = (int*) malloc(1000000);
    memcpy(temp_array, array, n*sizeof(int));
    array[i] = temp_array[y];
    array[y] = temp_array[i];
}

void bubbleSort(int array[], int n)
{
    int i,j;
    for(i = 0; i < n-1; i++)
    {
        for(j=0; j < n-i-1; j++)
        {
            if(array[j] > array[j+1])
            {
                sleep(1);
                swap(array, n, j, j+1);
            }
        }
    }
}

void printArray(int array[], int n)
{
    for (int i=0; i<n; i++)
    {
        printf("%d\n", array[i]);
    }
}

int main(int argc, char *argv[])
{
    int size = argc-1;

    int array[size];

    for (int x=0; x < size; x++)
    {
        array[x] = atoi(argv[x+1]);
    }

    bubbleSort(array,size);
    printArray(array,size);
}
```

Figure 1 *sorter.c*

La première chose qu'on peut apercevoir au premier coup d'œil, est qu'il n'y a pas de « free » à ce programme.

Avec des arguments différents :

```

$ valgrind ./sorter 8 12 1 3
==1144== Memcheck, a memory error detector
==1144== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1144== Using Valgrind-3.17.0 and LibVEX; rerun with -h for copyright info
==1144== Command: ./sorter 8 12 1 3
==1144==
Alloc
Alloc
Alloc
Alloc
[1
3
8
12
==1144==
==1144== HEAP SUMMARY:
==1144==   in use at exit: 400,000 bytes in 4 blocks
==1144==   total heap usage: 5 allocs, 1 frees, 401,024 bytes allocated
==1144==
==1144== LEAK SUMMARY:
==1144==   definitely lost: 400,000 bytes in 4 blocks
==1144==   indirectly lost: 0 bytes in 0 blocks
==1144==   possibly lost: 0 bytes in 0 blocks
==1144==   still reachable: 0 bytes in 0 blocks
==1144==   suppressed: 0 bytes in 0 blocks
==1144== Rerun with --leak-check=full to see details of leaked memory
==1144==
==1144== For lists of detected and suppressed errors, rerun with: -s
==1144== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

```

Figure 2 correction en labo

Ici, avec des arguments différents, on voit 5 allocations mémoire, mais seulement une libérée.

La fonction problématique est « swap », elle ne libère pas la mémoire.

On doit faire le free dans la fonction pour libérer la heap.

Valgrind est comme un débogueur. Il nous indique qu'il a perdu des bytes.

Si on le laisse tourner en boucle, il va prendre toute la mémoire, car elle n'est jamais libérée.

- Identifiez la source du problème dans le code source et corrigez le programme afin que la fuite mémoire ne se produise plus. Vérifiez à l'aide de l'analyse dynamique que le problème est bien réglé.

Corriger le problème.

Mettre un free pour éviter la saturation de la mémoire.

Code corrigé :

```

GNU nano 5.4
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

void swap(int array[], int n, int i, int y)
{
    int *temp_array = (int*) malloc(1000000);
    memcpy(temp_array, array, n*sizeof(int));
    array[i] = temp_array[y];
    array[y] = temp_array[i];
    free(temp_array);
}

void bubbleSort(int array[], int n)
{
    int i, j;
    for(i = 0; i < n-1; i++)
    {
        for(j=0; j < n-i-1; j++)
        {
            if(array[j] > array[j+1])
            {
                sleep(1);
                swap(array, n, j, j+1);
            }
        }
    }
}

void printArray(int array[], int n)
{
    for (int i=0; i<n; i++)
    {
        printf("%d\n", array[i]);
    }
}

int main(int argc, char *argv[])
{
    int size = argc-1;
    int array[size];

    for (int x=0; x < size; x++)
    {
        array[x] = atoi(argv[x+1]);
    }
}

```

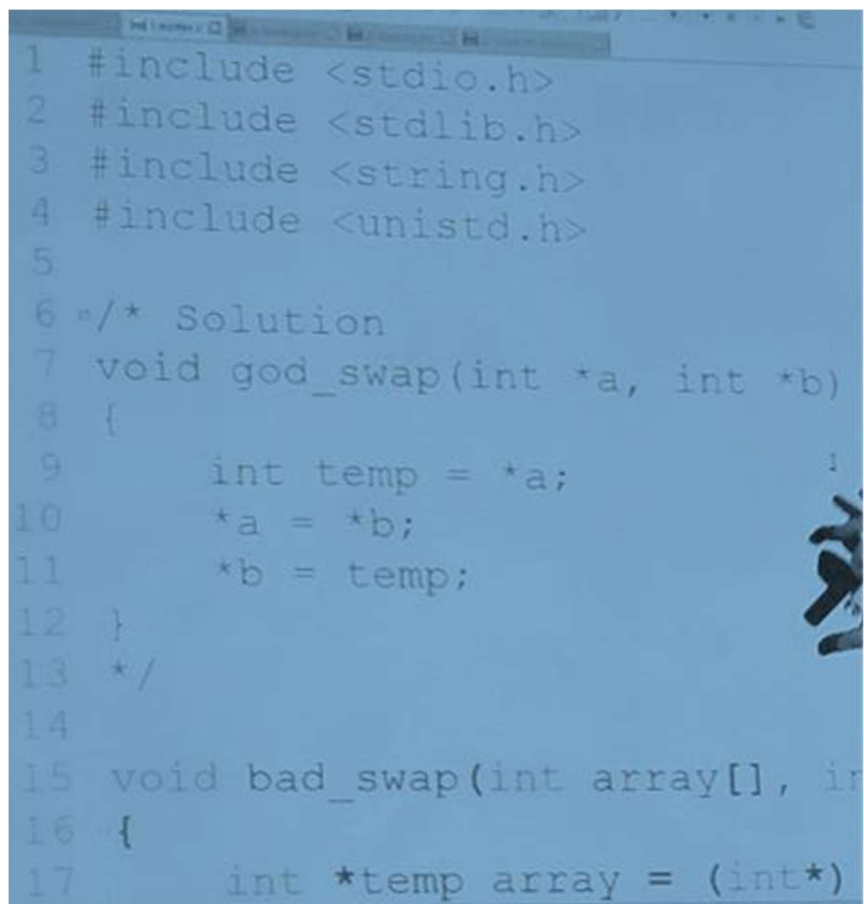
Figure 3 Figure 1 sorter.c corrigé

Résultat d'analyse :

```
(user@host)-[~/Documents/Labo 5/Lab5-20211121/ex1]
$ valgrind ./prog-1-corr razeraer azeraerazer azeraer azera zezraerazer 45454 qsfqsfsdf
=2167= Memcheck, a memory error detector
=2167= Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
=2167= Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
=2167= Command: ./prog-1-corr razeraer azeraerazer azeraer azera zezraerazer 45454 qsfqsfsdf
=2167=
0
0
0
0
0
0
45454
=2167=
=2167= HEAP SUMMARY:
=2167=   in use at exit: 0 bytes in 0 blocks
=2167=   total heap usage: 2 allocs, 2 frees, 101,024 bytes allocated
=2167=
=2167= All heap blocks were freed -- no leaks are possible
=2167=
=2167= For lists of detected and suppressed errors, rerun with: -s
=2167= ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Il n'y a plus d'erreur.

Une solution préférable :



```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5
6 /* Solution
7 void god_swap(int *a, int *b)
8 {
9     int temp = *a;
10    *a = *b;
11    *b = temp;
12 }
13 */
14
15 void bad_swap(int array[], int i, int j)
16 {
17     int *temp array = (int*)
```

Figure 4 sorter.c correction tableau

2 Fuzzing

La gestion des entrées dans un programme est souvent une partie délicate à gérer. En effet, un utilisateur bien ou mal intentionné pourra exploiter une faille si une entrée n'est pas correctement traitée par le développeur. Pour détecter ce genre de faille, la méthode la plus simple et la plus brutale reste le *fuzzing*. On va générer des entrées aléatoires afin d'identifier un éventuel bug.

Le programme est vulnérable, dans cet exercice, nous n'avons pas le code source.

On va envoyer plein d'input au programme et développer un script dont l'objectif est de trouver des bugs.

Il y a un exemple dans le dossier du labo 5 : « fuzzing_exemple ».

On va tester « custom_hash »

Ce programme est un système de hash.

2.1 Mise en situation (exercice à réaliser en groupe)

Un collègue de votre entreprise a développé une méthode *custom* pour générer des Hash (programme *custom_hash*). Il vous demande de tester son programme afin d'y trouver un éventuel bug. Développez un script Python permettant d'envoyer des chaînes aléatoires en entrée pour ce programme [3]. Si vous identifiez des problèmes, tentez de trouver une entrée qui reproduit systématiquement le bug.

Tests manuels avant :

```
(user@host)-[~/Documents/Labo 5/Lab5-20211121/ex2]
$ ./custom_hash bonjour
uzfcmzy

(user@host)-[~/Documents/Labo 5/Lab5-20211121/ex2]
$ ./custom_hash $
Traceback (most recent call last):
  File "custom_hash.py", line 28, in <module>
    ValueError: invalid literal for int() with base 10: '.'
[2458] Failed to execute script 'custom_hash' due to unhandled exception!
```

Lorsqu'un caractère spécial est entré, ce n'est pas supporté.

Test avec un script python :

```
GNU nano 5.4 fuzzing_example.py
import subprocess

with open ("file.txt","r") as file :
    for line in file :
        result = subprocess.run(["./custom_hash", line.rstrip()], stdout=subprocess.PIPE, stderr=subprocess.PIPE)
        #rstrip pour retirer tout ce qui est espace et retour à la ligne
        if result.stderr:
            print("Le caractere suivant fait crash le programme :")
            print(line)
```

Résultat :

```
(user@host)-[~/Documents/Labo 5/Lab5-20211121/ex2]
$ python3 fuzzing_example.py
Le caractere suivant fait crash le programme :
$
Le caractere suivant fait crash le programme :
=
Le caractere suivant fait crash le programme :
!
Le caractere suivant fait crash le programme :
*
Le caractere suivant fait crash le programme :
&
Le caractere suivant fait crash le programme :
é
```

...

Correction de l'exercice : que fais le programme ?

Il va nettoyer les entrées. Ensuite il divise les caractères par groupe de deux. Il appelle isalpha. Si ce n'est pas numérique, il fait une opération : il vérifie s'il n'y a pas de lettres. Mais il ne vérifie pas le point.

Une opération entière sur un chiffre et un point ne fonctionnera pas.

On ne peut pas faire un `int(.7)`. Vient s'ajouter à cela les caractères spéciaux qui font également planter le programme.

```
15
16 if len(sys.argv) != 2:
17     usage()
18
19 raw_input = re.sub(r"[^0-9a-zA-Z]+", "", sys.argv[1])
20 split_string(raw_input)
21
22 list_of_strings.reverse()
23 hash_list = []
24
25 for elem in list_of_strings:
26     if not any(c.isalpha() for c in elem):
27         # error here
28         hash_list.append(str(int(elem) % 11))
29     else:
30         new_string = ""
31         for c in elem:
32             new_string += chr((ord(c) + 1) % 26 + 97)
33         hash_list.append(new_string)
34
35 print(''.join(hash_list))
```

Figure 7 Code source de custom_hash

3 Tests unitaires

Le développement de logiciels est un processus collaboratif et qui est sujet à de nombreuses modifications. Le travail en équipe et ces modifications peuvent entraîner des bugs ou des régressions (bug introduit lors de la correction d'un autre bug ou de l'ajout d'une nouvelle fonctionnalité). Pour éviter ce genre d'erreurs, on utilise généralement des tests. Les tests unitaires vont par exemple permettre de tester fonction par fonction un programme, afin de vérifier que le comportement attendu est toujours le même après une modification.

3.1 Mise en situation (exercice à réaliser en groupe)

Vous êtes responsables de développer une série de tests pour un programme édité par votre entreprise (*calc.py*). Ce programme permet de réaliser les quatre opérations arithmétiques de base. On vous demande de créer une classe de test en utilisant le module *unittest*^[4] qui permettra de répondre aux contraintes définies par les PRE/POST^[5] de chaque fonction.

Avant de déployer un programme en production, on va lancer des tests.

Plusieurs façons d'y arriver : A la main ou test unitaire (test automatiser). Il va nous dire s'il y a des erreurs ou non.

Cela permet de vérifier la régression. C'est-à-dire, si quelque chose qui fonctionnait avant fonctionne toujours.

C'est utile en production avec les commites.

Si un programmeur modifie le programme et qu'il commet une erreur, ce type de programme nous préviendra.

Code donné :

```
class Calculator:

    # PRE: a and b are integers or floats
    # POST: the sum of a and b
    def sum(self, a, b):
        return a+b

    # PRE: a and b are integers or floats
    # POST: the result of the division of a and b. ZeroDivisionError if b=0
    def div(self, a, b):
        return a/b

    # PRE: a and b are integers or floats
    # POST: the result of the multiplication of a and b, in integer
    def mult(self, a, b):
        return int(a*b)

    # PRE: a and b are integers or floats
    # POST: the subtraction of a and b. If the result is less than 0, return 0
    def sub(self, a, b):
        if a-b < 0:
            return 0
        else:
            return a-b
```

Figure 8 calc.py

On préfix les fonctions par « test »

```
GNU nano 5.4
import unittest

from calc import Calculator

calcu = Calculator()

class TestStringMethods(unittest.TestCase):

    def test_add(self):
        self.assertEqual(calcu.sum(5,5),10)

    def test_new_div(self):
        self.assertEqual(calcu.div(10,2),5)

    def test_mlt(self):
        self.assertEqual(calcu.mult(5,5),25)

    def test_sub(self):
        self.assertEqual(calcu.sub(15,5),10)

if __name__ == '__main__':
    unittest.main()
```

4 Vérification des inputs en C

Comme vous avez pu le constater lors des exercices sur le *buffer-overflow*, un utilisateur peut augmenter ses privilèges en exploitant les inputs dans un programme légitime. Afin d'éviter cela, il est nécessaire de vérifier et de *nettoyer* ces inputs.

4.1 Mise en situation (exercice à réaliser en groupe)

Ecrivez un programme en C qui récupère un argument dans la console^[6], qui le stocke dans une variable pour ensuite l'imprimer à l'écran. Cet argument doit être d'une taille maximale de 10 et ne peut contenir que des caractères alphabétiques (pas de nombres, pas de caractères spéciaux). Testez votre programme en adaptant votre programme de *fuzzing* développé à l'exercice 2.

Vérification des inputs. Gérer tous les cas pour que le programme ne plante pas.

```

#define TAILLE 10

int isalphanum(char * arg, int n)
{
    for(int i=0;i<n;i++)
    {
        if(!isalpha(arg[i]))
        {
            return false;
        }
    }
    return true;
}

int main(int argc, char * argv[])
{
    char buffer[TAILLE];

    if(argc!=2)
    {
        printf("Entrer quelque chose et une seule chose");
        exit(EXIT_FAILURE);
    }
    else
    {
        printf("ok\n");

        if(strlen(argv[1]) > TAILLE)
        {
            printf("trop long!");
            exit(EXIT_FAILURE);
        } else {
            if(isalphanum(argv[1],strlen(argv[1])))
            {
                printf("%s est bien alphanumerique", argv[1]);
                strcpy(buffer, argv[1]);
                printf("Resultat: %s", buffer);
            }
            else
            {
                printf("please pour l'amour du ciel.... jpp de toi");
                exit(EXIT_FAILURE);
            }
        }
    }
}

```

On relance le programme de fuzzing et on ne voit pas d'erreur. Car le programme a été adapté pour ne pas en avoir. Il prend en compte tous les cas de figure d'une mauvaise utilisation de la part de l'utilisateur. Les erreurs sont anticipées par des messages d'avertissements et exit dans le programme.