

Programmation sécurisée



Projet : correction de failles de sécurités d'un
code écrit en C

Antoine Puissant

Enseignant : M. Dechaux

2014 - 2015

Résumé

L'objectif de ce projet est de corriger un code C. Ce dernier, donné par le professeur comporte des failles de sécurité qui pourraient être exploitées afin de réaliser des *buffer overflow* ou d'exécuter des codes injectés.

Table des matières

1	Introduction	3
1.1	Système de développement	3
1.2	Le code initial	3
1.3	Compilation du code	3
1.4	Exécution du code	6
2	Correction du code	8
2.1	Gestion du nombre d'arguments	8
2.2	Gestion des buffers	8
2.3	Premiers affichages	9
2.4	Réutilisation de <i>buf2</i>	10
2.5	Dernier affichage de <i>buf2</i>	10
2.5.1	Si <i>length</i> est supérieur à 0	11
2.5.2	Si <i>length</i> est inférieur ou égal à 0	11
2.6	Code final	12
3	Vérification de la correction	14
3.1	Tests du programme	14
3.2	Analyseur statique	14
3.3	Analyseur dynamique	15
4	Conclusion	18
	Table des figures	19

1 Introduction

1.1 Système de développement

Dans le cadre de ce projet, la correction a été réalisée sur un poste Linux¹. Ainsi, nous avons pu nous servir des outils tels que *Valgrind* ou *FlawFinder*. Les fonctions propres à *Microsoft* telles que *strcpy_s* n'ont ainsi pas été utilisées.

1.2 Le code initial

Dans un premier temps, il est nécessaire d'analyser le code original :

```
/* ----- */
/* Objectif : Corriger les erreurs statiques et dynamiques */
/*           de ce programme                               */
/* ----- */

int main(int argc, char *argv[])
{
    /* On ne touche pas a la variable "size" */
    int size = 16;
    int length = 0;
    char * buf;

    printf(argv[1]);

    buf = (char*)malloc(sizeof(char)*size + 1);

    strcpy_s(buf, sizeof(buf), argv[2]);
    printf("buffer : %s\n", buf);

    /* ----- */
    /* On ne touche pas au contenu "testdubuffer"          */
    /* mais on peut creer une variable reprenant ce contenu */
    /* ----- */

    strcpy_s(buf, sizeof(buf), "testdubuffer");

    length = strlen(buf);
    if(length != 0)
        printf("buffer : %s\n", buf);
    else
        free(buf);

    return 0;
}
```

1.3 Compilation du code

Afin de compiler ce code, nous remplacerons la fonction *strcpy_s* par la fonction *strcpy*.

Cependant, nous pouvons remarquer que ce code, ici écrit sous Visual Studio, ne réagit pas de la même façon une fois exécuté sous Windows. En effet, celui-ci va nous permettre de n'afficher que « *testdubu* ». Cela est dû à l'argument donnée à la fonction *strcpy_s* : *sizeof(buf)*. Ce dernier va alors faire une copie d'une taille de 8 octets (taille du pointeur). Ainsi, le fonctionnement basique

1. Ici, Ubuntu 14.04 LTS - 64 bits

du code est légèrement différent.

Nous utiliserons ensuite le fichier *Makefile* suivant afin de compiler le programme :

```

CC=gcc
CFLAGS=-W -Wall -ansi
EXEC=DynStat_old

all: $(EXEC)

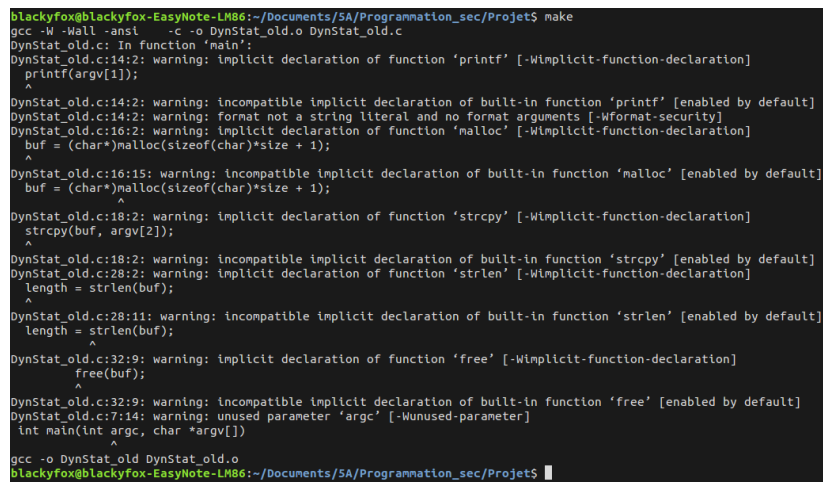
DynStat_old: DynStat_old.o
    $(CC) -o DynStat_old DynStat_old.o

DynStat.o_old: DynStat_old.c
    $(CC) -o DynStat_old.o -c DynStat_old.c

clean:
    rm -rf *.o

```

Lors de la première compilation, il est possible de se rendre compte que nous avons premièrement des *warning* :



```

blackbox@blackbox-EasyNote-LM86:~/Documents/SA/Programmation_sec/Projet$ make
gcc -W -Wall -ansi -c -o DynStat_old.o DynStat_old.c
DynStat_old.c: In function 'main':
DynStat_old.c:14:2: warning: implicit declaration of function 'printf' [-Wimplicit-function-declaration]
  printf(argv[1]);
  ^
DynStat_old.c:14:2: warning: incompatible implicit declaration of built-in function 'printf' [enabled by default]
DynStat_old.c:14:2: warning: format not a string literal and no format arguments [-Wformat-security]
DynStat_old.c:16:2: warning: implicit declaration of function 'malloc' [-Wimplicit-function-declaration]
  buf = (char*)malloc(sizeof(char)*size + 1);
  ^
DynStat_old.c:16:15: warning: incompatible implicit declaration of built-in function 'malloc' [enabled by default]
  buf = (char*)malloc(sizeof(char)*size + 1);
  ^
DynStat_old.c:18:2: warning: implicit declaration of function 'strcpy' [-Wimplicit-function-declaration]
  strcpy(buf, argv[2]);
  ^
DynStat_old.c:18:2: warning: incompatible implicit declaration of built-in function 'strcpy' [enabled by default]
DynStat_old.c:28:2: warning: implicit declaration of function 'strlen' [-Wimplicit-function-declaration]
  length = strlen(buf);
  ^
DynStat_old.c:28:11: warning: incompatible implicit declaration of built-in function 'strlen' [enabled by default]
  length = strlen(buf);
  ^
DynStat_old.c:32:9: warning: implicit declaration of function 'free' [-Wimplicit-function-declaration]
  free(buf);
  ^
DynStat_old.c:32:9: warning: incompatible implicit declaration of built-in function 'free' [enabled by default]
DynStat_old.c:7:14: warning: unused parameter 'argc' [-Wunused-parameter]
int main(int argc, char *argv[])
      ^
gcc -o DynStat_old DynStat_old.o
blackbox@blackbox-EasyNote-LM86:~/Documents/SA/Programmation_sec/Projet$

```

FIGURE 1 – Première compilation du programme

Ces erreurs sont du au fait que notre code n'a aucune bibliothèque d'incluse. Ainsi les fonctions utilisées, même les plus courantes comme *printf*, ne sont pas reconnues. Pour palier à cela, nous allons inclure trois bibliothèques :

```

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

```

Grâce à ces dernières, nous allons avoir accès à de nombreuses fonctions pour la gestion de chaînes de caractères, l'allocation dynamique de la mémoire, etc... Nous pouvons alors recompiler le programme et obtenir la sortie suivante :

```
blackyfox@blackyfox-EasyNote-LM86:~/Documents/SA/Programmation_sec/Projet$ make
gcc -W -Wall -ansi -c -o DynStat_old.o DynStat_old.c
DynStat_old.c: In function 'main':
DynStat_old.c:17:2: warning: format not a string literal and no format arguments [-Wformat-security]
    printf(argv[1]);
    ^
DynStat_old.c:10:14: warning: unused parameter 'argc' [-Wunused-parameter]
    int main(int argc, char *argv[])
            ^
gcc -o DynStat_old DynStat_old.o
blackyfox@blackyfox-EasyNote-LM86:~/Documents/SA/Programmation_sec/Projet$
```

FIGURE 2 – Seconde compilation du programme

Malgré les *warnings*, un fichier exécutable a été créé. Nous pouvons alors tester le programme.

1.4 Exécution du code

En l'exécutant sans aucun argument donné, nous obtenons la sortie suivante :

```
blackyfox@blackyfox-EasyNote-LM86:~/Documents/SA/Programmation_sec/Projet$ ./DynStat_old
buffer : XDG_VTNR=7
buffer : testdubuffer
blackyfox@blackyfox-EasyNote-LM86:~/Documents/SA/Programmation_sec/Projet$
```

FIGURE 3 – Première exécution du programme

On se rend alors compte que nous avons déjà ici des erreurs. En effet, en lisant le code, nous pouvons voir que le programme récupère deux chaînes de caractères données en arguments. Ici, bien qu'aucun argument n'ai été donné, le programme s'exécute. On peut aussi remarquer sur la première ligne de la sortie (c.f. figure ??) que le programme affiche une valeur. Cette dernière, censée être le second argument donné, est en réalité une variable d'environnement du système.

En donnant un argument au programme nous obtenons le résultat suivant :

```
blackyfox@blackyfox-EasyNote-LM86:~/Documents/SA/Programmation_sec/Projet$ ./DynStat_old argument
Erreur de segmentation (core dumped)
blackyfox@blackyfox-EasyNote-LM86:~/Documents/SA/Programmation_sec/Projet$
```

FIGURE 4 – Seconde exécution du programme

Ici, nous avons une erreur de segmentation. Nous pouvons utiliser l'outil *Valgrind* afin de comprendre un peu plus pourquoi nous avons eu cette erreur :

```

blackyfox@blackyfox-EasyNote-LM86:~/Documents/SA/Programmation_sec/Projet$ valgrind ./DynStat_old argument
==11748== Memcheck, a memory error detector
==11748== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==11748== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==11748== Command: ./DynStat_old argument
==11748==
==11748== Invalid read of size 1
==11748==    at 0x4C2E1C7: strcpy (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==11748==    by 0x4006C0: main (in /home/blackyfox/Documents/SA/Programmation_sec/Projet/DynStat_old)
==11748== Address 0x0 is not stack'd, malloc'd or (recently) free'd
==11748==
==11748== Process terminating with default action of signal 11 (SIGSEGV)
==11748== Access not within mapped region at address 0x0
==11748==    at 0x4C2E1C7: strcpy (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==11748==    by 0x4006C0: main (in /home/blackyfox/Documents/SA/Programmation_sec/Projet/DynStat_old)
==11748== If you believe this happened as a result of a stack
==11748== overflow in your program's main thread (unlikely but
==11748== possible), you can try to increase the size of the
==11748== main thread stack using the --main-stacksize= flag.
==11748== The main thread stack size used in this run was 8388608.
argument==11748==
==11748== HEAP SUMMARY:
==11748==    in use at exit: 17 bytes in 1 blocks
==11748== total heap usage: 1 allocs, 0 frees, 17 bytes allocated
==11748==
==11748== LEAK SUMMARY:
==11748==    definitely lost: 0 bytes in 0 blocks
==11748==    indirectly lost: 0 bytes in 0 blocks
==11748==    possibly lost: 0 bytes in 0 blocks
==11748==    still reachable: 17 bytes in 1 blocks
==11748==    suppressed: 0 bytes in 0 blocks
==11748== Rerun with --leak-check=full to see details of leaked memory
==11748==
==11748== For counts of detected and suppressed errors, rerun with: -v
==11748== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
Erreur de segmentation (core dumped)
blackyfox@blackyfox-EasyNote-LM86:~/Documents/SA/Programmation_sec/Projet$

```

FIGURE 5 – Seconde exécution du programme avec *Valgrind*

Ainsi, nous pouvons voir que le programme a essayé de lire un espace mémoire qui ne lui était pas réservé.

Nous savons donc que nous devons faire attention au nombre d'arguments donnés au programme lors de la correction.

Enfin, en donnant deux arguments au programme nous obtenons la sortie suivante :

```

blackyfox@blackyfox-EasyNote-LM86:~/Documents/SA/Programmation_sec/Projet$ ./DynStat_old argument second
argumentbuffer : second
buffer : testdubuffer
blackyfox@blackyfox-EasyNote-LM86:~/Documents/SA/Programmation_sec/Projet$

```

FIGURE 6 – Troisième exécution du programme

Dans ce cas, nous pouvons voir que le programme réalise bien ce qu'il lui a été demandé :

1. Afficher le premier argument
2. Afficher « buffer : » suivi du contenu de la variable *buf*, contenant le second argument
3. Afficher « buffer : » suivi du contenu de la variable *buf*, contenant la chaîne « testdubuffer »

Si nous donnons des arguments supplémentaires au programme, nous voyons que ce dernier ne traite pas ces arguments :

```

blackyfox@blackyfox-EasyNote-LM86:~/Documents/SA/Programmation_sec/Projet$ ./DynStat_old argument second troisieme
argumentbuffer : second
buffer : testdubuffer
blackyfox@blackyfox-EasyNote-LM86:~/Documents/SA/Programmation_sec/Projet$

```

FIGURE 7 – Quatrième exécution du programme

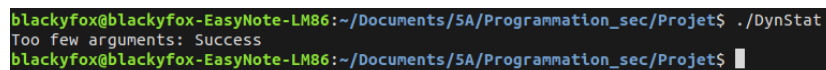
2 Correction du code

2.1 Gestion du nombre d'arguments

Dans un premier temps, nous allons mettre en place un système permettant de traiter le cas où le nombre d'arguments n'est pas suffisant. Pour ce faire, nous allons faire un test sur la variable `argc` :

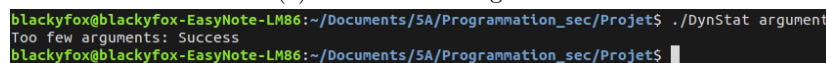
```
/* Checking if we have only two arguments given */
if (argc == 3)
{
    ...
    /* code */
    ...
}
else{
    if (argc < 3)
    {
        /* If there are less than two arguments given, */
        /* exiting with error code 1 */
        perror("Too few arguments");
        return 1;
    }
    else{
        /* If there are more than two arguments given, */
        /* exiting with error code 2 */
        perror("Too many arguments");
        return 2;
    }
}
```

Ainsi, si nous exécutons le code en n'ayant pas deux arguments, nous obtiendrons les résultats suivants :



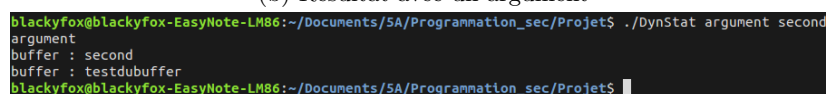
```
blackyfox@blackyfox-EasyNote-LM86:~/Documents/5A/Programmation_sec/Projet$ ./DynStat
Too few arguments: Success
blackyfox@blackyfox-EasyNote-LM86:~/Documents/5A/Programmation_sec/Projet$
```

(a) Résultat sans argument



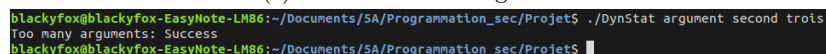
```
blackyfox@blackyfox-EasyNote-LM86:~/Documents/5A/Programmation_sec/Projet$ ./DynStat argument
Too few arguments: Success
blackyfox@blackyfox-EasyNote-LM86:~/Documents/5A/Programmation_sec/Projet$
```

(b) Résultat avec un argument



```
blackyfox@blackyfox-EasyNote-LM86:~/Documents/5A/Programmation_sec/Projet$ ./DynStat argument second
argument
buffer : second
buffer : testdubuffer
blackyfox@blackyfox-EasyNote-LM86:~/Documents/5A/Programmation_sec/Projet$
```

(c) Résultat avec 2 arguments



```
blackyfox@blackyfox-EasyNote-LM86:~/Documents/5A/Programmation_sec/Projet$ ./DynStat argument second trois
Too many arguments: Success
blackyfox@blackyfox-EasyNote-LM86:~/Documents/5A/Programmation_sec/Projet$
```

(d) Résultat avec 3 d'arguments ou plus

FIGURE 8 – Résultats en fonction du nombre d'arguments

2.2 Gestion des buffers

Comme nous avons pu le voir précédemment, nous devons ici récupérer deux chaînes de caractères données en arguments à notre programme. Il est alors intéressant d'utiliser deux buffers pour stocker ces deux variables.

Nous allons donc dans un premier temps créer ces variables :

```
|| char* buf1 = NULL;  
|| char* buf2 = NULL;
```

Nous allons ensuite allouer dynamiquement l'espace mémoire de ces variables. Nous limiterons leur taille grâce à la variable *size*.

Pour ce faire, nous utilisons la fonction *calloc*. Cette fonction est intéressante pour nous car en plus de permettre une allocation dynamique de la mémoire comme la fonction *malloc*, elle va permettre de remplir cet espace mémoire de 0. Ainsi, aucune corruption dues aux données présentes à cet espace précédemment ne pourrons interférer avec notre programme.

```
|| buf1 = (char*)calloc(strlen(argv[1], size)+1, sizeof(char));  
|| buf2 = (char*)calloc(strlen(argv[2], size)+1, sizeof(char));
```

Ainsi, la taille allouée correspond à la taille de l'argument, majorée par la variable *size*².

Ensuite, nous allons récupérer les chaînes de caractères données en arguments et stocker dans les buffers prévus à cet effet. Pour cela, nous allons utiliser la fonction *snprintf*. Cette dernière va pouvoir nous permettre de copier les contenus d'*argv[1]* et d'*argv[2]* de manière beaucoup contrôlée. En effet, cette fonction va nous permettre de vérifier si la taille des données n'est pas supérieure à la taille du buffer d'arrivée contrairement à la fonction *strcpy*. Cela va ainsi nous permettre de limiter la taille des données traitées afin d'empêcher un *buffer overflow*.

La copie se fait alors de la manière suivante :

```
|| if(snprintf(buf1, strlen(argv[1], size)+1, "%s", argv[1]) < 0){  
||     perror("Writing on variable failed");  
||     return 5;  
|| }  
|| if(snprintf(buf2, strlen(argv[2], size)+1, "%s", argv[2]) <  
||     0){  
||     perror("Writing on variable failed");  
||     return 6;  
|| }
```

De plus nous plaçons cette fonction dans un test conditionnel afin de vérifier que son déroulement s'est exécuté correctement.

2.3 Premiers affichages

Comme nous avons pu le voir dans la partie 1.4, lors de l'exécution du code, le programme va, dans un premier temps, afficher le premier argument puis le second. Ces derniers étant maintenant stockés dans des variables (*buf1* et *buf2*), nous allons pouvoir les afficher simplement et de manière plus sécurisée. Comme nous avons pu le voir durant la seconde session de travaux pratiques de ce cours, il est possible de réaliser des attaques de chaînes formatées³ lorsque nous utilisons la fonction *printf* de la forme suivante :

```
|| printf(argv[1]);
```

Ainsi, nous allons ici utiliser la fonction *printf* de manière plus sécurisée :

```
|| printf("%s\n", buf1);  
|| printf("buffer : %s\n", buf2);
```

2. Ici *size* = 16

3. Format String attacks

2.4 Réutilisation de *buf2*

En regardant attentivement le code source d'origine, nous pouvons constater que la variable *buf2* se fait « écraser » afin de contenir la chaîne de caractères suivante : « testdubuffer ».

Afin de nous faciliter la tâche et d'être sûr que nous utiliserons bien cette chaîne de caractères, nous allons la sauvegarder dans une variable que nous allouerons dynamiquement :

```
char* testdubuffer = NULL;
...
testdubuffer = (char*)calloc(strlen("testdubuffer", 12)+1,
    sizeof(char));
...
if(snprintf(testdubuffer, strlen("testdubuffer", 12)+1, "%s", "
testdubuffer") < 0){
    perror("Writing on variable failed");
    return 7;
}
```

Ainsi, nous avons alloué un espace mémoire ne pouvant contenir que 12 caractères, soit les 12 de la chaîne cible.

Afin de copier le contenu de cette variable dans notre buffer (*buf2*), nous allons tout d'abord libérer l'espace mémoire occupé par la précédente allocation mémoire. Nous allons ensuite ré-allouer de la mémoire afin de stocker le contenu de la variable *testdubuffer*. Enfin, nous allons copier le contenu de la variable *testdubuffer* dans notre buffer cible :

```
free(buf2);
buf2 = NULL;
buf2 = (char*)calloc(strlen(testdubuffer, 12)+1, sizeof(char));

if(snprintf(buf2, strlen(testdubuffer, 12)+1, "%s",
    testdubuffer) < 0){
    perror("Writing on variable failed");
    return 4;
}
```

Ainsi, la variable *buf2* va contenir le contenu de la variable *testdubuffer* et à une allocation mémoire limitée à 12 caractères.

Ensuite, nous récupérerons la taille de notre buffer modifié. Pour cela, nous n'utilisons pas la fonction *strlen* mais une version plus sécurisée de celle-ci, *strlen*. Cette dernière va pouvoir nous permettre de majorer la taille voulue. Ici, elle sera de 12.

```
length = strlen(buf2, 12)+1;
```

Il est possible de remarquer que nous avons rajouté un +1 à la fin de l'assignation de la taille. En effet, les fonctions *strlen* et *strlen* ne prennent pas en compte le caractère de fin de chaîne lors du comptage. Ainsi, nous devons rajouter ce +1 afin d'être sûr de copier la chaîne correctement.

2.5 Dernier affichage de *buf2*

Avant de pouvoir afficher le contenu de notre buffer, nous allons vérifier que ce dernier n'est pas vide et qu'il contient bien la chaîne de caractères désirés.

Pour ce faire, nous allons réaliser un test conditionnel sur la variable *length* contenant la taille de notre buffer.

2.5.1 Si *length* est supérieur à 0

Dans ce cas, nous pouvons assumer que notre buffer contient bel et bien notre variable. Nous pouvons donc l'afficher.

Nous faisons donc un simple appel à la fonction *printf* :

```
|| printf("buffer : %s\n", buf2);
```

Une fois ce dernier effectué, nous devons libérer toute la mémoire allouée. Cela correspond aux variables :

- *buf1*
- *buf2*
- *testdubuffer*

Pour ce faire, nous utilisons la fonction *free* :

```
|| free(buf2);  
|| free(buf1);  
|| free(testdubuffer);
```

Enfin pour être certain que les données ne soient plus accessibles, nous allons assigner à nos variable la valeur de *NULL*. Ainsi, elles ne pointeront plus sur l'espace mémoire qu'elles occupaient au préalable.

```
|| buf2 = NULL;  
|| buf1 = NULL;  
|| testdubuffer = NULL;
```

Enfin, nous allons réassigner les valeurs des autres variables pour que celles-ci ne soient plus accessibles non plus :

```
|| size = 0;  
|| length = 0;
```

Enfin, nous pouvons retourner 0 car l'exécution du programme c'est déroulée correctement.

```
|| return 0;
```

2.5.2 Si *length* est inférieur ou égal à 0

Dans ce cas, nous pouvons assumer que la copie de la variable *testdubuffer* dans *buf2* a échouée. Ainsi, nous émettons un message d'erreur :

```
|| perror("Copy of testdubuffer to buf2 failed");
```

Cependant, nous ne quittons le programme tout de suite. Comme expliqué dans le point 2.5.1, nous avons alloué de la mémoire précédemment dans le code. Il est alors important de la libérer. Nous allons aussi affecter des valeurs *NULL* ou 0 à nos variables :

```
|| free(buf2);  
|| free(buf1);  
|| free(testdubuffer);  
|| buf2 = NULL;  
|| buf1 = NULL;  
|| testdubuffer = NULL;  
|| size = 0;  
|| length = 0;
```

Enfin, nous avons rencontré une erreur alors nous quittons le programme avec un code d'erreur, ici le code 3 :

```
|| return 3;
```

2.6 Code final

Ainsi, une fois corrigé, il est possible de retrouver le code suivant :

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

/* ----- */
/* Objectif : Corriger les erreurs statiques et dynamiques */
/*           de ce programme */
/* ----- */

/* Correction made by Antoine Puissant */

int main(int argc, char *argv[])
{
    /* Creation of the variables and initialization */
    int size = 16;
    int length = 0;
    char* buf1 = NULL;
    char* buf2 = NULL;
    char* testdubuffer = NULL;

    /* Checking if we have only two arguments given */
    if (argc == 3)
    {
        /* Memory allocation for the two args given */
        /* Allocation for the size of the argument (max = size = 16) */
        buf1 = (char*)calloc(strlen(argv[1], size)+1, sizeof(char))
        ;
        buf2 = (char*)calloc(strlen(argv[2], size)+1, sizeof(char))
        ;

        /* Memory allocation for the "testdubuffer" variable */
        testdubuffer = (char*)calloc(strlen("testdubuffer", 12)+1,
            sizeof(char));

        /* Writing argv[1] in buf1, argv[2] in buf2 and "
            testdubuffer" in testdubuffer */
        if(snprintf(buf1, strlen(argv[1], size)+1, "%s", argv[1]) <
            0){
            perror("Writing on variable failed");
            return 5;
        }
        if(snprintf(buf2, strlen(argv[2], size)+1, "%s", argv[2]) <
            0){
            perror("Writing on variable failed");
            return 6;
        }
        if(snprintf(testdubuffer, strlen("testdubuffer", 12)+1, "%s
            ", "testdubuffer") < 0){
            perror("Writing on variable failed");
            return 7;
        }
    }
}
```

```
/* Outputting buf1, then buf2 */
printf("%s\n", buf1);
printf("buffer : %s\n", buf2);

/* Memory liberation of buf2. Then reallocation for the
   right size */
free(buf2);
buf2 = NULL;
buf2 = (char*)calloc(strlen(testdubuffer, 12)+1, sizeof(
    char));

/* Writing testdubuffer in buf2 */
if(snprintf(buf2, strlen(testdubuffer, 12)+1, "%s",
    testdubuffer) < 0){
    perror("Writing on variable failed");
    return 4;
}

/* Getting size of buf2 */
length = strlen(buf2, 12)+1;

/* If writing worked (size>0), outputting buf2 and free
   memory */
if(length > 0){
    printf("buffer : %s\n", buf2);
    /* If writing failed, free memory and exit with error code 3
       */
}else{
    perror("Copy of testdubuffer to buf2 failed");
    ret = 3;
}
free(buf2);
free(buf1);
free(testdubuffer);
buf2 = NULL;
buf1 = NULL;
testdubuffer = NULL;
size = 0;
length = 0;
return ret;
}else{
    if (argc < 3)
    {
        /* If there are less than two arguments given, exiting
           with error code 1 */
        perror("Too few arguments");
        return 1;
    }else{
        /* If there are more than two arguments given, exiting
           with error code 2 */
        perror("Too many arguments");
        return 2;
    }
}
}
```

3 Vérification de la correction

Afin de vérifier si la correction appliquée est correcte, nous allons procéder en trois étapes.

3.1 Tests du programme

La première partie de vérification est celle de l'utilisation du programme. Il est en effet possible de tester le programme afin de voir si tous les cas imaginables par l'utilisateur sont pris en compte.

3.2 Analyseur statique

Un analyseur statique est un programme qui va analyser le code source d'un autre programme. Celui-ci va ainsi donner des conseils quand à l'utilisation de certaines fonctions au sein du code.

Ici, nous utiliserons l'analyseur statique *FlawFinder*.

En l'utilisant sur le code de base, nous obtenons la sortie suivante :

```
blackyfox@blackyfox-EasyNote-LM86:~/Documents/5A/Programmation_sec/Projet$ flawfinder DynStat_old.c
Flawfinder version 1.27, (C) 2001-2004 David A. Wheeler.
Number of dangerous functions in C/C++ ruleset: 160
Examining DynStat_old.c
DynStat_old.c:17: [4] (format) printf:
  If format strings can be influenced by an attacker, they can be
  exploited. Use a constant for the format specification.
DynStat_old.c:21: [4] (buffer) strcpy:
  Does not check for buffer overflows when copying to destination.
  Consider using strncpy or strlcpy (warning, strncpy is easily misused).
DynStat_old.c:29: [2] (buffer) strcpy:
  Does not check for buffer overflows when copying to destination.
  Consider using strncpy or strlcpy (warning, strncpy is easily misused). Risk
  is low because the source is a constant string.
DynStat_old.c:31: [1] (buffer) strlen:
  Does not handle strings that are not \0-terminated (it could cause a
  crash if unprotected).

Hits = 4
Lines analyzed = 37 in 0.51 seconds (3724 lines/second)
Physical Source Lines of Code (SLOC) = 38
Hits@level = [0] 0 [1] 1 [2] 1 [3] 0 [4] 2 [5] 0
Hits@level+ = [0+] 4 [1+] 4 [2+] 3 [3+] 2 [4+] 2 [5+] 0
Hits/KSLOC@level+ = [0+] 105.263 [1+] 105.263 [2+] 78.9474 [3+] 52.6316 [4+] 52.6316 [5+] 0
Minimum risk level = 1
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!
blackyfox@blackyfox-EasyNote-LM86:~/Documents/5A/Programmation_sec/Projet$
```

FIGURE 9 – Résultat de *FlawFinder* sur le code de base

Comme nous pouvons le voir plusieurs fonctions sont à examiner de plus près d'après *FlawFinder* :

1. La fonction *printf* ligne 17. Cette dernière prend en argument seulement *argv[1]* ce qui peut permettre à des attaquants d'utiliser du code formaté pour détourner l'utilisation du programme.
2. La fonction *strcpy* lignes 21 et 29. Cette fonction ne vérifie pas si le buffer de destination est assez grand pour accueillir le contenu du buffer d'arrivée. Il y a alors un risque de *buffer overflow*.
3. La fonction *strlen* ligne 31. Comme expliqué précédemment (c.f. 2.4), cette fonction ne gère pas le caractère de fin de chaîne et peut ainsi faire planter le programme en cas de mauvaise gestion.

En utilisant *FlawFinder* sur notre code corrigé, nous obtenons le résultat suivant :

```
blackyfox@blackyfox-EasyNote-LM86:~/Documents/SA/Programmation_sec/Projet$ flawfinder DynStat.c
Flawfinder version 1.27, (C) 2001-2004 David A. Wheeler.
Number of dangerous functions in C/C++ ruleset: 160
Examining DynStat.c
DynStat.c:34:  [1] (port) snprintf:
  On some very old systems, snprintf is incorrectly implemented and
  permits buffer overflows; there are also incompatible standard definitions
  of it. Check it during installation, or use something else.
DynStat.c:38:  [1] (port) snprintf:
  On some very old systems, snprintf is incorrectly implemented and
  permits buffer overflows; there are also incompatible standard definitions
  of it. Check it during installation, or use something else.
DynStat.c:42:  [1] (port) snprintf:
  On some very old systems, snprintf is incorrectly implemented and
  permits buffer overflows; there are also incompatible standard definitions
  of it. Check it during installation, or use something else.
DynStat.c:57:  [1] (port) snprintf:
  On some very old systems, snprintf is incorrectly implemented and
  permits buffer overflows; there are also incompatible standard definitions
  of it. Check it during installation, or use something else.

Hits = 4
Lines analyzed = 101 in 0.51 seconds (8319 lines/second)
Physical Source Lines of Code (SLOC) = 102
Hits@Level = [0]  0 [1]  4 [2]  0 [3]  0 [4]  0 [5]  0
Hits@Level+ = [0+]  4 [1+]  4 [2+]  0 [3+]  0 [4+]  0 [5+]  0
Hits/KSLOC@Level+ = [0+] 39.2157 [1+] 39.2157 [2+]  0 [3+]  0 [4+]  0 [5+]  0
Minimum risk level = 1
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!
blackyfox@blackyfox-EasyNote-LM86:~/Documents/SA/Programmation_sec/Projet$
```

FIGURE 10 – Résultat de *FlawFinder* sur le code corrigé

Ici, nous pouvons clairement voir qu’une seule fonction n’est remarquée par *FlawFinder* : *snprintf* lignes 34, 38, 42 et 57.

L’analyseur nous indique que dans de vieux systèmes cette fonction était mal implémentée et qu’il faut alors se prémunir de certains risques pour ce genre de systèmes. Dans notre cas, cela n’a que peu d’importance. En effet, en ayant utilisé des tests conditionnels autour des valeurs de retour de cette fonction, nous nous protégeons de potentielles erreurs.

Ainsi, nous pouvons dire que notre code corrigé est valide selon *FlawFinder*.

3.3 Analyseur dynamique

Cependant, comme expliqué dans le point précédent, un analyseur statique ne s’occupe que du code source et non de l’exécution du programme. Ainsi, pour vérifier s’il n’y a pas de fuite mémoire dans notre code, nous utiliserons l’outil *Valgrind*.

Nous allons pouvoir comparer les sorties de *Valgrind* sur le programme d’origine et celui corrigé.


```

==17186== Invalid read of size 1
==17186== at 0x480071c: 10_file_expungeq2libc_2.5 (fileops.c:1362)
==17186== by 0x4829294: vfprintf (vfprintf.c:1663)
==17186== by 0x4800401: printf (printf.c:33)
==17186== by 0x4000001: main (in /home/blackfox/Documents/SA/Programmation_sec/Projet/dynstat_old)
==17186== Address 0x51f0c5d is 12 bytes after a block of size 17 alloc'd
==17186== at 0x4000001: malloc (in /usr/lib/valgrind/valgrind_mencheck-amd64-linux.so)
==17186== by 0x4000002: main (in /home/blackfox/Documents/SA/Programmation_sec/Projet/dynstat_old)
==17186==
==17186== Invalid read of size 8
==17186== at 0x4813773: 42_memcpy (memcpy.c:1322)
==17186== by 0x4800401: 10_file_expungeq2libc_2.5 (fileops.c:1320)
==17186== by 0x4829294: vfprintf (vfprintf.c:1663)
==17186== by 0x4800401: printf (printf.c:33)
==17186== by 0x4000001: main (in /home/blackfox/Documents/SA/Programmation_sec/Projet/dynstat_old)
==17186== Address 0x51f0c5f is 15 bytes inside a block of size 17 alloc'd
==17186== at 0x4000001: malloc (in /usr/lib/valgrind/valgrind_mencheck-amd64-linux.so)
==17186== by 0x4000002: main (in /home/blackfox/Documents/SA/Programmation_sec/Projet/dynstat_old)
==17186==
==17186== Invalid read of size 8
==17186== at 0x4813773: 42_memcpy (memcpy.c:1323)
==17186== by 0x4800401: 10_file_expungeq2libc_2.5 (fileops.c:1320)
==17186== by 0x4829294: vfprintf (vfprintf.c:1663)
==17186== by 0x4800401: printf (printf.c:33)
==17186== by 0x4000001: main (in /home/blackfox/Documents/SA/Programmation_sec/Projet/dynstat_old)
==17186== Address 0x51f0c5f is 8 bytes after a block of size 17 alloc'd
==17186== at 0x4000001: malloc (in /usr/lib/valgrind/valgrind_mencheck-amd64-linux.so)
==17186== by 0x4000002: main (in /home/blackfox/Documents/SA/Programmation_sec/Projet/dynstat_old)
==17186==
buffer : (EISDIR) /usr/bin/lsnpipe No
buffer : testsubuffer
==17186== in use at exit: 17 bytes in 1 blocks
==17186== total heap usage: 1 allocs, 0 frees, 17 bytes allocated
==17186==
==17186== LEAK SUMMARY:
==17186==    definitely lost: 17 bytes in 1 blocks
==17186==    indirectly lost: 0 bytes in 0 blocks
==17186==    possibly lost: 0 bytes in 0 blocks
==17186==    still reachable: 0 bytes in 0 blocks
==17186==    suppressed: 0 bytes in 0 blocks
==17186== Rerun with --leak-check=full to see details of leaked memory
==17186==
==17186== For counts of detected and suppressed errors, rerun with: -v
==17186== ERROR SUMMARY: 46 errors from 7 contexts (suppressed: 0 from 0)
blackfox@blackfox-aspire-1480:~/Documents/SA/Programmation_sec/Projet$

```

(a) Résultat de *Valgrind* pour le code original sans argument

```

blackfox@blackfox-aspire-1480:~/Documents/SA/Programmation_sec/Projet$ valgrind ./dynstat
==17293== Memcheck, a memory error detector
==17293== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==17293== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==17293== Command: ./dynstat
==17293==
too few arguments: Success
==17293==
==17293== HEAP SUMMARY:
==17293==    in use at exit: 0 bytes in 0 blocks
==17293==    total heap usage: 1 allocs, 1 frees, 568 bytes allocated
==17293==
==17293== All heap blocks were freed -- no leaks are possible
==17293==
==17293== For counts of detected and suppressed errors, rerun with: -v
==17293== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
blackfox@blackfox-aspire-1480:~/Documents/SA/Programmation_sec/Projet$

```

(b) Résultat de *Valgrind* pour le code corrigé sans argumentFIGURE 11 – Résultat de *Valgrind* pour une exécution sans argument

```

blackfox@blackfox-aspire-1480:~/Documents/SA/Programmation_sec/Projet$ valgrind ./dynstat_old premier
==17286== Memcheck, a memory error detector
==17286== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==17286== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==17286== Command: ./dynstat_old premier
==17286==
==17286== Invalid read of size 1
==17286== at 0x481c1c7: strcpy (in /usr/lib/valgrind/valgrind_mencheck-amd64-linux.so)
==17286== by 0x4000001: main (in /home/blackfox/Documents/SA/Programmation_sec/Projet/dynstat_old)
==17286== Address 0x5 is not stack'd, malloc'd or (recently) free'd
==17286==
==17286== Process terminating with default action of signal 11 (SIGSEGV)
==17286== Access not within mapped region at address 0x5
==17286== at 0x481c1c7: strcpy (in /usr/lib/valgrind/valgrind_mencheck-amd64-linux.so)
==17286== by 0x4000001: main (in /home/blackfox/Documents/SA/Programmation_sec/Projet/dynstat_old)
==17286== If you believe this happened as a result of a bug
==17286== overflow in your program's main thread (unlikely but
==17286== possible), you can try to increase the size of the
==17286== main thread stack using the --main-stacksize=flag.
==17286== The main thread stack size used in this run was 819200.
premier==17286==
==17286== HEAP SUMMARY:
==17286==    in use at exit: 17 bytes in 1 blocks
==17286==    total heap usage: 1 allocs, 0 frees, 17 bytes allocated
==17286==
==17286== LEAK SUMMARY:
==17286==    definitely lost: 0 bytes in 0 blocks
==17286==    indirectly lost: 0 bytes in 0 blocks
==17286==    possibly lost: 17 bytes in 1 blocks
==17286==    still reachable: 0 bytes in 0 blocks
==17286==    suppressed: 0 bytes in 0 blocks
==17286== Rerun with --leak-check=full to see details of leaked memory
==17286==
==17286== For counts of detected and suppressed errors, rerun with: -v
==17286== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
Erreur de segmentation (core dumped)
blackfox@blackfox-aspire-1480:~/Documents/SA/Programmation_sec/Projet$

```

(a) Résultat de *Valgrind* pour le code original avec un argument

```

blackfox@blackfox-aspire-1480:~/Documents/SA/Programmation_sec/Projet$ valgrind ./dynstat premier
==17312== Memcheck, a memory error detector
==17312== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==17312== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==17312== Command: ./dynstat premier
==17312==
too few arguments: Success
==17312==
==17312== HEAP SUMMARY:
==17312==    in use at exit: 0 bytes in 0 blocks
==17312==    total heap usage: 1 allocs, 1 frees, 568 bytes allocated
==17312==
==17312== All heap blocks were freed -- no leaks are possible
==17312==
==17312== For counts of detected and suppressed errors, rerun with: -v
==17312== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
blackfox@blackfox-aspire-1480:~/Documents/SA/Programmation_sec/Projet$

```

(b) Résultat de *Valgrind* pour le code corrigé avec un argumentFIGURE 12 – Résultat de *Valgrind* pour une exécution avec un argument

```

blackfox@blackfox-aspire-1480:~/Documents/SA/Programmation_sec/Projet$ valgrind ./dynstat_old premier second
==17254== Memcheck, a memory error detector
==17254== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==17254== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==17254== Command: ./dynstat_old premier second
==17254==
buffer: second
buffer : testsubuffer
==17254== in use at exit: 17 bytes in 1 blocks
==17254== total heap usage: 1 allocs, 0 frees, 17 bytes allocated
==17254==
==17254== LEAK SUMMARY:
==17254==    definitely lost: 17 bytes in 1 blocks
==17254==    indirectly lost: 0 bytes in 0 blocks
==17254==    possibly lost: 0 bytes in 0 blocks
==17254==    still reachable: 0 bytes in 0 blocks
==17254==    suppressed: 0 bytes in 0 blocks
==17254== Rerun with --leak-check=full to see details of leaked memory
==17254==
==17254== For counts of detected and suppressed errors, rerun with: -v
==17254== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
blackfox@blackfox-aspire-1480:~/Documents/SA/Programmation_sec/Projet$

```

(a) Résultat de *Valgrind* pour le code original avec deux arguments

```

blackfox@blackfox-aspire-1480:~/Documents/SA/Programmation_sec/Projet$ valgrind ./dynstat premier second
==17272== Memcheck, a memory error detector
==17272== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==17272== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==17272== Command: ./dynstat premier second
==17272==
buffer : second
buffer : testsubuffer
==17272==
==17272== HEAP SUMMARY:
==17272==    in use at exit: 0 bytes in 0 blocks
==17272==    total heap usage: 1 allocs, 4 frees, 41 bytes allocated
==17272==
==17272== All heap blocks were freed -- no leaks are possible
==17272==
==17272== For counts of detected and suppressed errors, rerun with: -v
==17272== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
blackfox@blackfox-aspire-1480:~/Documents/SA/Programmation_sec/Projet$

```

(b) Résultat de *Valgrind* pour le code corrigé avec deux argumentsFIGURE 13 – Résultat de *Valgrind* pour une exécution avec deux arguments

[illegible]

FIGURE 14 – Résultat de *Valgrind* pour une exécution avec trois arguments

Ainsi, nous pouvons conclure que ce code est sécurisée du point de vue de l'analyseur dynamique.

4 Conclusion

Nous avons pu voir durant ce projet qu'à partir d'un code relativement simple, nous pouvons trouver de nombreuses failles de sécurité. Après une analyse approfondie du code, de son fonctionnement et des fonctions utilisées, il est possible de le corriger de manière efficace.

L'utilisation d'outils permettant d'analyser le code de manière statique ou dynamique est d'une grande aide dans ce genre de projet et permet de mieux comprendre le fonctionnement de certaines fonctions ainsi que leurs avantages et inconvénients.

Table des figures

1	Première compilation du programme	5
2	Seconde compilation du programme	6
3	Première exécution du programme	6
4	Seconde exécution du programme	6
5	Seconde exécution du programme avec <i>Valgrind</i>	7
6	Troisième exécution du programme	7
7	Quatrième exécution du programme	7
8	Résultats en fonction du nombre d'arguments	8
9	Résultat de <i>FlawFinder</i> sur le code de base	14
10	Résultat de <i>FlawFinder</i> sur le code corrigé	15
11	Résultat de <i>Valgrind</i> pour une exécution sans argument	16
12	Résultat de <i>Valgrind</i> pour une exécution avec un argument	16
13	Résultat de <i>Valgrind</i> pour une exécution avec deux arguments	16
14	Résultat de <i>Valgrind</i> pour une exécution avec trois arguments	17