

Laboratoire 02 - 03 - 04 :
Uboot - Compile Kernel - Valgrind

Département : EIE
Unité d'enseignement : MA_SeS

Auteurs : Quentin Müller & Tristan Traiber
Professeur : Jean-Roland Schuler
Classe : C2 - C3
Date : 8 avril 2022

Table des matières

1	Introduction	3
2	Laboratoire Uboot	3
2.1	Question 1 : Configuration du u-boot	3
2.2	Question 2 : Modifier la partition du boot en ext4	4
2.3	Question 3 : Changer l'initialisation réseau	5
2.4	Question 4 : -fstack-protector-all gcc option	7
3	Laboratoire Kernel configuration	9
3.1	Question 1 : Configuration d'un Kernel sécurisé	9
3.1.1	Compléments d'informations sur l'activation des paquets . . .	9
3.1.2	Configuration d'un Kernel sécurisé	10
3.1.3	Désactiver quelques options du Kernel	12
3.1.4	Nécessaire pour la suite des laboratoires	12
3.2	Question 2 : Amélioration de la sécurité du Kernel lors du démarrage	12
3.3	Initialisation de <i>/etc/sysctl.conf</i>	12
3.4	Écrire le script d'initialisation	13
4	Laboratoire Valgrind	14
4.1	Question 1-1 : Analyse de l'outil Cachegrind	14
4.2	Question 1-2 : Analyse de l'outil Massif	17
4.3	Question 1-3 : Analyse de l'outil Memcheck	21
4.3.1	Essai d'accès mémoire depuis une autre pile	21
4.3.2	Essai d'utilisation d'une variable non initialisée	22
4.3.3	Essai de la double libération de la mémoire	23
4.3.4	Essai de copie avec "memcpy"	24
4.3.5	Essai de l'allocation dynamique sans libération	25
4.4	Question 2 : Analyse d'un code avec les outils de Valgrind	26
5	Conclusion	29
6	Signatures	29
7	Annexes	30
A	Codes	30
A.1	Fichier <i>bitmap_corr.c</i>	31

1 Introduction

L'objectif de cette première série de laboratoires est d'appréhender les problèmes de sécurité liés aux systèmes embarqués et plus particulièrement au système d'exploitation de Linux. Le but est de premièrement prendre en main les outils et l'environnement et d'expérimenter trois thèmes différents. Une première partie s'intéresse à l'initialisation du système en abordant Uboot. Puis une deuxième partie permet d'expérimenter la consolidation de la sécurité du noyau Linux en abordant le Kernel. Puis finalement, la troisième partie permet d'expérimenter plusieurs outils de Valgrind et de mettre en pratique ces fonctionnalités sur un cas concret.

2 Laboratoire Uboot

La première partie de ce rapport est sur u-boot pour *Universal – Boot* est un logiciel libre, utilisé comme chargeur d'amorçage, surtout sur les systèmes embarqués. Ce qui signifie que lors du lancement du système d'exploitation d'un système embarqué l'u-boot va démarrer le kernel linux.

2.1 Question 1 : Configuration du u-boot

La première question de ce laboratoire consiste à modifier le prompt par défaut du u-boot. Pour ce faire, il est nécessaire de se diriger sur la VM dans le dossier `home/lmi/workspace/nano/buildroot/` puis d'ouvrir un terminal à cet endroit afin de pouvoir taper la commande suivante :

`make uboot-menuconfig`

Une interface pseudo-graphique s'ouvre. Dans cette interface, il faut trouver le menu *command line interface* puis *shell prompt*. Il est alors possible de modifier le prompt `=>` par `Nano Pi #`. Afin de valider les changements, il est nécessaire de recompiler uboot. Pour ce faire, il faut utiliser la commande `make uboot-rebuild`.

Une fois le changement effectué, il est nécessaire de flasher à nouveau la carte microSD. Une fois cette étape réalisée, il est possible de contrôler le changement effectué en se connectant au nanopi avec la commande `minicom`. Lorsque que minicom démarre, il faut taper une touche du clavier dans un intervalle de 2 secondes afin de démarrer uboot. Il est alors possible de constater que le changement à bien été configuré, comme démontré dans la figure 1 :

A screenshot of a terminal window titled 'lmi@fedora:~ — minicom'. The terminal output shows the u-boot boot process: 'scanning bus usb@1c1d000 for devices... 1 USB Device(s) found', 'scanning bus usb@1c1d400 for devices... 1 USB Device(s) found', 'scanning usb for storage devices... 0 Storage Device(s) found', 'Hit any key to stop autoboot: 0', and finally the prompt 'Nano Pi #'. The terminal has a dark background and a scrollbar on the right side.

```
lmi@fedora:~ — minicom
scanning bus usb@1c1d000 for devices... 1 USB Device(s) found
scanning bus usb@1c1d400 for devices... 1 USB Device(s) found
scanning usb for storage devices... 0 Storage Device(s) found
Hit any key to stop autoboot: 0
Nano Pi #
```

FIGURE 1 – Prompt de uboot du nanoPI

2.2 Question 2 : Modifier la partition du boot en ext4

Durant le premier laboratoire, nous avons dû écrire un script permettant de flasher la carte microSD avec deux partitions. Une partition **vfat** de **64MB** nommée *BOOT* et une partition **ext4** de **2GB** nommée *rootfs*. Pour ce faire nous avons donc écrit le script *generate.sh* permettant de réaliser cette tâche. Pour la suite des laboratoires, il nous est demandé de modifier ce script afin de modifier la partition **vfat** *BOOT* en ext4.

Cette partie se réalise en quatre étapes.

1. Dans le script *generate.sh*, modifier les lignes 27 et 31 en remplaçant **fat** par **ext4**
2. Dans le dossier `workspace/nano/buildroot/board/friendlyarm/nanopi-neo-plus2` avec la commande `nano boot.cmd` modifier le fichier en remplaçant les **fatload** par **ext4load**
3. Recréer le fichier `boot.src` avec la commande `mkimage -C none -A arm64 -T script -d board/friendlyarm/nanopi-neo-plus2/boot.cmd /home/lmi /workspace/nano/buildroot/output/images/boot.scr` dans le dossier `buildroot`
4. Utiliser le script *generate.sh* pour flasher le carte microSD

Le script *generate.sh* est décrit dans le listing 1

```
1 #!/bin/sh
2 # These 3 variables must be modified according to user setup
3 # They are not subject to change (check SD root folder just in case
4 # Default values according to course slides (1_nanopi.pdf)
5 SD_ROOT_FOLDER=/dev/sdb          # SD card root directory, unmounted
6 LOCAL_MNT_POINT=/run/media/lmi  #default SD card mount point
7 HOME_DIR=/home/lmi              # user home directory
8
9 SD_ROOT_PART1=${SD_ROOT_FOLDER}1
10 SD_ROOT_PART2=${SD_ROOT_FOLDER}2
11
12 umount $SD_ROOT_FOLDER
13 #initialize 480MiB to 0
14 sudo dd if=/dev/zero of=$SD_ROOT_FOLDER count=120000
15 sync
16
17 # First sector: msdos
18 sudo parted $SD_ROOT_FOLDER mklabel msdos
19
20 #copy sunxi-spl.bin binaries
21 sudo dd if=${HOME_DIR}/workspace/nano/buildroot/output/images/sunxi-
    -spl.bin of=$SD_ROOT_FOLDER bs=512 seek=16
22
23 #copy u-boot
24 sudo dd if=${HOME_DIR}/workspace/nano/buildroot/output/images/u-
    boot.itb of=$SD_ROOT_FOLDER bs=512 seek=80
25
26 # 1st partition: 64MiB: (163840-32768)*512/1024 = 64MiB
27 sudo parted $SD_ROOT_FOLDER mkpart primary ext4 32768s 163839s
28
```

```
29 # 2nd partition: 1GiB: 4358144-163840)*512/1024 = 2GiB
30 sudo parted $SD_ROOT_FOLDER mkpart primary ext4 163840s 4358143s
31 sudo mkfs.ext4 $SD_ROOT_PART2 -L rootfs
32 sudo mkfs.ext4 $SD_ROOT_PART1 -L BOOT
33 sync
34
35 #copy kernel, flattened device tree, boot.scr
36 sudo mount $SD_ROOT_PART1 $LOCAL_MNT_POINT
37 sudo cp ${HOME_DIR}/workspace/nano/buildroot/output/images/Image
   $LOCAL_MNT_POINT
38 sudo cp ${HOME_DIR}/workspace/nano/buildroot/output/images/nanopi-
   neo-plus2.dtb $LOCAL_MNT_POINT
39 sudo cp ${HOME_DIR}/workspace/nano/buildroot/output/images/boot.scr
   $LOCAL_MNT_POINT
40 sync
41
42 sudo dd if=${HOME_DIR}/workspace/nano/buildroot/output/images/
   rootfs.ext4 of=$SD_ROOT_PART2
43 # Resize and rename 2nd partition to rootfs
44 # check if the partition must be mounted
45 sudo e2fsck -f $SD_ROOT_PART2
46 sudo resize2fs $SD_ROOT_PART2
47 sudo e2label $SD_ROOT_PART2 rootfs
```

Listing 1 – Listing du script generate.sh

Le fait de changer le script generate.sh permet de modifier la façon dont la partition *BOOT* est créée. Au lieu de définir une partition **fat**, on définit une partition **ext4** tout comme la deuxième partition *ROOTFS*. La commande **mkimage** est utilisée pour créer les images utilisées avec le chargeur de démarrage U-Boot. Ces images peuvent contenir le noyau Linux, l'arborescence des périphériques, l'image du système de fichiers racine, les images du micrologiciel, etc.

2.3 Question 3 : Changer l'initialisation réseau

L'objectif de cette partie du laboratoire est de modifier l'initialisation réseau du nanoPI. Il est possible de réaliser cette modification directement sur le nanoPI mais cette solution a pour défaut que cette étape doit être répétée à chaque fois que la carte microSD est flashée avec une nouvelle configuration.

Pour que le fichier soit généré à chaque fois qu'on exécute la commande **make**, c'est-à-dire que l'on recompile, nous pouvons placer la configuration réseaux dans le dossier `buildroot/board/friendlyarm/nanopi-neo-plus2/rootfs_overlay`. Cela car le contenu de `rootfs_overlay/` est copié dans `buildroot/output/target/` lors du démarrage du nanoPI.

Pour réaliser cette modification, il faut respecter les étapes suivantes :

1. Dans le dossier `buildroot/board/friendlyarm/nanopi-neo-plus2/rootfs_overlay`
2. Créer un répertoire *etc* avec la commande `mkdir etc`
3. Entrer dans le dossier avec la commande `cd etc`
4. Créer un répertoire *network* avec la commande `mkdir network`
5. Entrer dans le dossier avec la commande `cd network`
6. Créer un fichier *interfaces* avec la commande `nano interfaces`
7. Finalement, remplir le fichier *interfaces* comme décrit dans le listing 2

```
1 auto eth0
2 iface eth0 inet static
3 address 192.168.0.11
4 netmask 255.255.255.255
5 gateway 192.168.0.4
6 pointopoint 192.168.0.4
7
8 auto lo
9 iface lo inet loopback
```

Listing 2 – Contenu du fichier *interfaces*

La deuxième partie est pour l'adresse de loopback cette partie étant déjà présente dans le fichier du nanopi, nous l'avons rajoutée à la fin pour être sûr de ne pas perdre des fonctionnalités.

Pour vérifier que notre modification est bien valide, nous avons flasher la carte microSD avec le même script que pour la deuxième partie puis lancé la commande `ifconfig` directement sur le nanoPI afin d'obtenir les informations de la configuration réseau.

La figure 2, illustre le résultat de la manipulation décrite ci-dessus.

```
Welcome to FriendlyARM Nanopi NEO Plus2
buildroot login: root
# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:01:AD:86:CA:19
          inet addr:192.168.0.11  Bcast:0.0.0.0  Mask:255.255.255.255
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Interrupt:22

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

FIGURE 2 – Résultat de la commande *ifconfig* sur la nanoPI

2.4 Question 4 : -fstack-protector-all gcc option

Il est possible, à l'aide d'un code C qui réalise un buffer overflow et un buffer underflow d'écraser des données qui se trouve dans la stack et ainsi pouvoir corrompre le contenu de la stack. Il existe cependant une option `-fstack-protector-all` du compilateur `gcc` qui permet de contrer ce type de comportement. Le principe est simple. Le compilateur place un canari entre la mémoire alloué au programme et le reste de la stack. Lorsque l'on réalise un dépassement de mémoire (buffer overflow), on va écraser ce canari avant d'écraser la stack. Lorsque ce canari est écrasé, le compilateur arrête le programme et retourne une erreur.

La dernière partie de ce laboratoire permet de tester l'efficacité du `-fstack-protector-all` du compilateur `gcc`. Pour faire cela, nous avons écrit un petit programme en C qui réalise un buffer overflow et un buffer underflow pour garantir de tomber sur le cas où l'on va écrase le canari.

Afin de pouvoir tester le programme sur la nanoPI, nous avons réalisé une cross-compilation de notre programme C avec la commande `aarch64-none-linux-gnu-gcc` puis, nous avons placé le fichier `.out` de sortie dans le dossier `rootfs/root` de la carte microSD du nanoPI. Le code permettant le buffer overflow et le buffer underflow est illustré par la figure 3.

```
1  ~/*
2  Laboratoire de SeS 2 sur u-boot
3  Quentin Müller et Tristan Traiber
4  30.10.2021
5  */
6
7  #include <stdio.h>
8
9  #define SIZE_TAB 10
10 #define SIZE_OVERFLOW 30
11 #define SIZE_UNDERFLOW 30
12
13 int main() {
14     printf("SeS - Labo u-boot 'Quentin Müller et Tristan Traiber'\n\n");
15
16     char tab[SIZE_TAB] = {0};
17
18     printf("Le tableau va se remplir de 0 à N-1 puis déborder sur la stack de %d octets ", SIZE_OVERFLOW);
19
20     printf("\nDébut du programme\n");
21
22     for (char i = 0; i < SIZE_TAB + SIZE_OVERFLOW; i++) {
23         tab[i] = i;
24         //printf("%d, ", tab[i]);
25     }
26
27     printf("\n Fin de l'écriture plus overflow\n");
28
29     for (char i = 0; i < SIZE_UNDERFLOW; i++) {
30         tab[-i] = i;
31         //printf("%d, ", tab[-i]);
32     }
33     printf("\n Fin de l'écriture plus overflow\n");
34
35     printf("Fin du programme\n\n");
36     return 0;
37 }
38 }
```

FIGURE 3 – Code C du test du stack protector

La figure 4, illustre le résultat obtenu.

```
[lmi@fedora soft_question4]$ ./comp.sh
[lmi@fedora soft_question4]$ ./main
SeS - Labo u-boot 'Quentin Müller et Tristan Traiber'

Le tableau va se remplir de 0 à N-1 puis deborder sur la stack de 30 octets
Début du programme
[]
Fin de l'écriture plus overflow
[]
Fin de l'écriture plus overflow
Fin du programme

Segmentation fault (core dumped)
[lmi@fedora soft_question4]$ ./main_protected
SeS - Labo u-boot 'Quentin Müller et Tristan Traiber'

Le tableau va se remplir de 0 à N-1 puis deborder sur la stack de 30 octets
Début du programme
[]
Fin de l'écriture plus overflow
[]
Fin de l'écriture plus overflow
Fin du programme

*** stack smashing detected ***: terminated
Aborted (core dumped)
```

FIGURE 4 – Test de la détection du buffer overflow

Sur la figure 4 nous constatons deux comportements, premièrement nous avons lancé le programme sans l'option de stack protector afin de voir les effets. Puis, nous avons lancé le même programme avec l'option stack protector. Sur la deuxième partie nous constatons que le programme crache avec le message d'erreur **** stack smashing detected ****. Le protecteur de stack a donc bien détecté le buffer overflow.

3 Laboratoire Kernel configuration

L'objectif de ce laboratoire est d'aborder les questions relatives à la sécurité du noyau (Kernel) Linux.

La réalisation est décomposée en deux parties :

1. Configuration d'un Kernel sécurisé
2. Amélioration de la sécurité du Kernel lors du démarrage

3.1 Question 1 : Configuration d'un Kernel sécurisé

Dans cette partie, nous allons réaliser la configuration d'un Kernel Linux sécurisé. Nous précisons que lorsque nous demandons **d'activer** une option cela se traduit par `[*]` dans le menu de configuration. Au contraire lorsque nous demandons de **désactiver** une option cela se traduit par `[]`.

3.1.1 Compléments d'informations sur l'activation des paquets

Lors de ce laboratoire, nous avons été amené à devoir activer des options du Kernel Linux. Sur certains de ces options trois possibilités s'offraient à nous `[*]`, `<M>`, `[]`. Comme expliqué auparavant, afin d'activer une options ou modules nous allons mettre `[*]` et quand nous voulons la désactiver `[]`. Il reste donc une options qui n'a pas été décrite `<M>`. Cette configuration n'étant pas disponible pour chaque option, nous avons commencé par chercher quel type d'option offre cette possibilité et ce que cela signifie. Nous avons donc trouver que seule les modules peuvent avoir cette annotation `<M>` et que cela peut être vu comme l'exprime la figure 5.

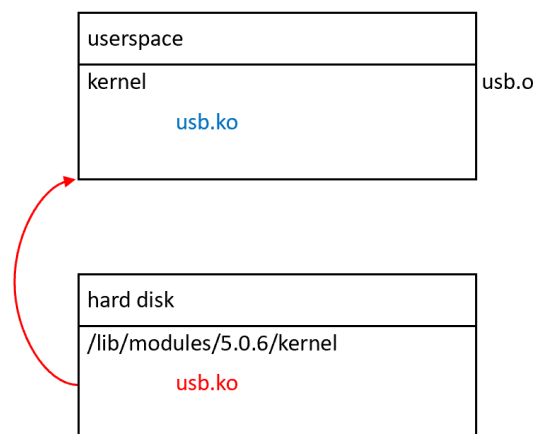


FIGURE 5 – Différence entre activation `<M>` et `[*]`

Sur cette figure, nous pouvons observer deux lieux de stockage des modules, soit directement dans le userspace, soit dans la carte microSD dans notre cas. Si l'on choisit le `<*>` Alors nous nous trouvons dans le cas de figure en bleu ou le module

se trouve directement dans le Kernel du userspace. Cela signifie quand cas de besoin de cette ressource, le Kernel pourra directement y avoir accès. Si l'on choisit l'option `<M>` alors nous nous trouvons dans le cas rouge ou le module est stocké sur la carte microSD et au moment de l'utilisation de ce module il est chargé dans le userspace. Cette configuration offre des avantages et des inconvénient. L'avantage principales est la libération de l'espace mémoire du userspace. Par contre en cas de besoin fréquent de cette ressource, il va y avoir un temps de chargement du module qui va ralentir le début de l'exécution. Donc cela signifie qu'il faut choisir assez méthodiquement les modules à mettre en externe dans la carte microSD.

La configuration que nous avons réalisée respecte la marche à suivre suivante :

3.1.2 Configuration d'un Kernel sécurisé

1. Se diriger dans le dossier `~/workspace/nano/buildroot` puis exécuter la commande `make linux-menuconfig` pour ouvrir l'interface semi-graphique de configuration de linux
2. Dans le répertoire *Platform selection*, **désactiver tout sauf** :
 - *Allwinner sunxi 64-bit SoC Family*
 - *Broadcom BCM2835 family*
3. Dans le répertoire *General Setup* :
 - **Désactiver** *Kernel config support*
 - **Désactiver** *Disable Heap randomization*
 - Dans le sous dossier *Choose SLAB allocator (SLAB)* **choisir** *SLAB*
 - **Activer** *Allow slab caches to be merged*
 - **Activer** *SLAB freelist randomization*
 - Dans le sous dossier *Compiler optimization level*
 - **Choisir** *Optimize for performance*
4. Dans le répertoire *General architecture-dependent options*
 - **Activer** *Stack Protector buffer overflow detection*
 - **Activer** *Strong Stack Protector*
5. Dans le répertoire *Kernel feature*
 - **Activer** *Randomize the address of the kernel image*
 - **Activer** *Apply r/o permissions of VM areas also to their linear aliases*
6. Dans le répertoire *Device drivers*
 - Dans le sous dossier *Character Devices*
 - **Activer** *Hardware Random Number Generator Core support*
 - Dans le sous dossier *Hardware random number generator core support*
 - **Activer** *Timer IOMEM HW Random Number Generator support*
 - **Activer** *Broadcom BCM2835/BCM63xx Random Number Generator support*

7. Dans le répertoire *Kernel Hacking*
 - **Activer** *Filter access to /dev/mem*
 - **Activer** *Filter I/O access to /dev/mem*
 - Dans le sous dossier *Compile time check and compiler options*
 - **Désactiver** *Compile the kernel with debug info*
 - **Activer** *Strip assembler-generated symbols during link*
8. Dans le répertoire *Security options*
 - **Activer** *Restrict unprivileged access to the kernel syslog*
 - **Activer** *Harden memory copies between kernel and userspace*
 - **Activer** *Allow usercopy whitelist violations to fallback to object size*
 - **Activer** *Harden common str/mem functions against buffer overflows*
 - Dans le sous dossier du chemin *Kernel Hardening Options/Memory Initialization/Initialize Kernel Stack ...*
 - **Activer** *Enable heap memory zeroing on allocation by default*
 - **Activer** *Enable heap memory zeroing on free by default*
9. Dans le sous dossier *Networking options* du répertoire *Networking support*
 - **Activer** *TCP IP Networking*
 - **Activer** la nouvelle option *IP : TCP syncookie support activer*
10. Dans le répertoire *File Systems* :
 - Respecter la configuration de la figure 6

```

make linux-xconfig: File Systems →
(For each file system you use, make sure extended attribute support is enabled)
<*> Second extended fs support
    [*]    Ext2 extended attributes
    [*]    Ext2 POSIX Access Control Lists
    [*]    Ext2 Security Labels

<*> The extended 3 (ext3) file system
    [*]    Ext3 POSIX Access Control Lists
    [*]    Ext3 Security Labels

<*> The Extended 4 (ext4) file system
    [*]    Ext4 POSIX Access Control Lists
    [*]    Ext4 Security Labels

```

FIGURE 6 – Configuration à respecter pour le répertoire *File Systems*

3.1.3 Désactiver quelques options du Kernel

Si l'on souhaite avoir une vue d'ensemble de toutes les options activées dans le Kernel Linux, il est possible de trouver la liste complète dans le fichier `.config`. Pour cela il faut se placer dans `\workspace\nano\buildroot\output\build\linux-x.x.x`. Une fois le bon répertoire trouvé, il est possible d'afficher le fichier `.config` avec la commande `less .config`. Il est ensuite possible de naviguer dans ce fichier et de réaliser des recherches afin de contrôler l'état de la configuration du Kernel Linux. En suite, il est possible désactiver des options du Kernel Linux à l'aide de la commande `make linux-menuconfig` comme mentionné dans les étapes précédentes.

Dans notre cas, la configuration du Kernel nous semblait satisfaisante et nous n'avons pas trouvé d'option supplémentaire ou inutile à désactiver. Dans la donnée de laboratoire, un des exemples qui a été donné, était de désactiver le CD-ROM, mais cette option est déjà désactivée dans la version du Kernel Linux que nous utilisons.

3.1.4 Nécessaire pour la suite des laboratoires

En prévisions des prochains laboratoires, il est nécessaire de réaliser une dernière modification :

1. Dans le répertoire *General Setup* de `linux-menuconfig`
 - **Activer** *Initial RAM File system and RAM Disk*

Une fois toutes les modifications réalisées, il est encore nécessaire de compiler le Kernel avec la commande `make`.

Attention : cette opération prend beaucoup de temps, il faut donc s'assurer d'avoir bien respecté toutes les étapes. Une fois terminé, il faut contrôler qu'aucune erreur n'est apparu durant la compilation.

3.2 Question 2 : Amélioration de la sécurité du Kernel lors du démarrage

L'objectif de cette question est d'améliorer la sécurité du kernel Linux lors du démarrage. Pour commencer, nous désactivons le mode routeur du nanoPI. Pour réaliser cela, il faut taper la commande `sysctl -w net.ipv4.ip_forward=0` dans le terminal du nanoPI. Il est possible de vérifier que la commande a fonctionné en tapant la commande `cat /proc/sys/net/ipv4/ip_forward`. L'exécution de cette commande doit retourner `0`.

3.3 Initialisation de `/etc/sysctl.conf`

Dans `[breakline=true]workspace/nano/buildroot/board/friendlyarm/nanopi-neo-plus2/rootfs-overlay/etc`. Nous devons créer le fichier `sysctl.conf`.

Attention : Si l'on exécute la commande `sudo nano sysctl.conf` on le crée un fichier en mode super user ce qui change notre emplacement `lmi` par `root` ce qui bloque son utilisation par le système après coup. Il faut donc faire attention à simplement exécuter la commande `nano sysctl.conf` pour créer le fichier avec l'utilisateur (`lmi`).

Il faut donc créer le fichier *sysctl.conf* avec la commande `nano sysctl.conf`, puis y inscrire les informations données dans la slide 39 du fichier *3_complieKernel* du cours *SeS*. Le listing 3 décrit le contenu à ajouter au fichier *sysctl.conf* :

```
1 kernel.randomize_va_space=2
2 net.ipv4.conf.lo.rp_filter=1
3 net.ipv4.conf.eth0.rp_filter=1
4 net.ipv4.conf.lo.accept_source_route=0
5 net.ipv4.conf.eth0.accept_source_route=0
6 net.ipv4.conf.lo.accept_redirects=0
7 net.ipv4.conf.eth0.accept_redirects=0
8 net.ipv4.icmp_echo_ignore_broadcasts=1
9 net.ipv4.icmp_ignore_bogus_error_responses=1
10 net.ipv4.conf.lo.log_martians=1
11 net.ipv4.conf.eth0.log_martians=1
12 net.ipv4.tcp_syncookies=1
```

Listing 3 – Contenu de *sysctl.conf*

3.4 Écrire le script d'initialisation

Une fois que le fichier *sysctl.conf* a été créé et complété. Il faut encore qu'il puisse être exécuté lors du démarrage du nanoPI. Pour cela il est nécessaire de créer un script d'initialisation.

La marche à suivre pour créer le script d'initialisation est la suivante :

Dans *workspace/nano/buildroot/board/friendlyarm/nanopi-neo-plus2/rootfs-overlay/etc*.

1. Créer le répertoire *init.d* avec la commande `mkdir init.d`
2. Entrer dans ce répertoire avec la commande `cd init.d`
3. Créer le fichier *S00KernelParameter* avec la commande `nano S00KernelParameter`
4. Écrire dans le fichier *S00KernelParameter* la commande `sysctl -p`
5. Sauvegarder et quitter le fichier
6. Dans le dossier *workspace/nano/buildroot*
 - Compiler avec la commande `make`
7. Flasher la carte en utilisant le script *generate.sh*

Il est ensuite possible de contrôler que la configuration fonctionne bien en vérifiant le contenu de *sysctl.conf* directement sur le nanoPI. Ce fichier se trouve dans */proc/sys/net/ipv4/*.

4 Laboratoire Valgrind

L'objectif de ce laboratoire est d'aborder la question des optimisations de programmation aussi bien pour des fuites de mémoire "memory leak" ou limiter son nombre d'accès. Les fuites de mémoires sont souvent dues à des erreurs de programmation et peuvent être difficiles à diagnostiquer. Grâce à Valgrind et à ces outils, il nous est plus facile d'identifier la source de ces erreurs, ainsi que les solutions d'optimisation du code afin réduire son impact sur la mémoire. Ce laboratoire est divisé en deux parties.

1. Test des différents outils à disposition avec Valgrind
2. Utilisation des outils pour trouver les erreurs d'un programme C

4.1 Question 1-1 : Analyse de l'outil Cachegrind

L'outil cachegrind sert à vérifier l'utilisation de la mémoire cache. Pour ce faire, nous avons réalisé deux programmes, un bon qui servira de référence et un mauvais de même base, mais avec une différence de code qui normalement engendre une augmentation du nombre d'accès mémoire.

Sur le listing 4, nous pouvons observer l'exemple de bon programme. Il est composé une double boucle `for` pour accéder à chaque case d'un tableau à deux dimensions. Il est important de noter que nous parcourons la dimension la plus faible du tableau (`j`) dans la boucle `for` interne.

```
1 | for (int i = 0; i < N; i++)  
2 | {  
3 |     for (int j = 0; j < N; j++)  
4 |     {  
5 |         tableau[i][j] = 1;  
6 |     }  
7 | }
```

Listing 4 – Code du bon exemple

La figure 7 illustre le résultat obtenu avec l'outil cachegrind pour l'exemple de bon programme.

D	refs:	5,734,227	(4,538,110 rd	+ 1,196,117 wr)
D1	misses:	31,698	(13,579 rd	+ 18,119 wr)
LLd	misses:	24,752	(7,543 rd	+ 17,209 wr)
D1	miss rate:	0.6%	(0.3%	+ 1.5%)
LLd	miss rate:	0.4%	(0.2%	+ 1.4%)

FIGURE 7 – Capture d'écran du log de sortie de cachegrind avec un bon programme

Les valeurs importantes à relever sur la figure 7 sont **D1 misses** et **D1 miss rate**. Nous constatons que pour l'exemple de bon programme, la valeur de **D1 misses** vaut 31,698 et que le **D1 miss rate** vaut 0.6%.

Nous réalisons maintenant la même expérience, mais avec l'exemple de mauvais programme. Comme nous pouvons le voir sur listing 5, la seule différence est l'inversion des deux indices du tableau parcouru par les deux boucles **for**. Ceci peu sembler anodin, mais cela a un impact important sur les accès mémoires.

```

1 | for (int i = 0; i < N; i++)
2 | {
3 |     for (int j = 0; j < N; j++)
4 |     {
5 |         tableau[j][i] = 1; //i <-> j
6 |     }
7 | }
```

Listing 5 – Code du mauvais exemple

D	refs:	5,734,229	(4,538,112 rd	+ 1,196,117 wr)
D1	misses:	1,016,057	(13,572 rd	+ 1,002,485 wr)
LLd	misses:	24,752	(7,547 rd	+ 17,205 wr)
D1	miss rate:	17.7%	(0.3%	+ 83.8%)
LLd	miss rate:	0.4%	(0.2%	+ 1.4%)

FIGURE 8 – Capture d'écran du log de sortie de cachegrind avec un mauvais programme

Si nous comparons la figure 7 et la figure 8, nous pouvons observer une très importante augmentation de la rubrique D1 misses. Cette rubrique représente le nombre d'accès mémoire inutile. Ce nombre passe de 31'698 à 1'016'057 soit 32 fois plus accès.

Ceci s'explique de manière assez simple si l'on regarde l'architecture des données en mémoire et leur accès comme illustré dans la figure 9.

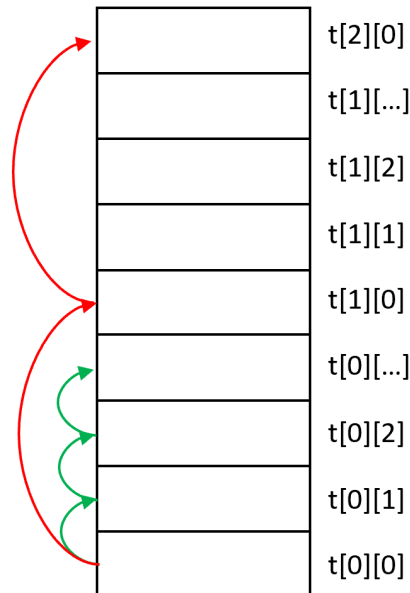


FIGURE 9 – Représentation des données en mémoire

Les données du tableau à deux dimensions que nous avons déclarées sont stockées dans la mémoire de manière consécutive en suivant l'indice de la dimension la plus faible, soit ce qui est représenté par la paire de crochet `[]` la plus à droite.

Le comportement du bon programme est représenté par les flèches vertes. Nous constatons que dans ce cas, les accès au donné du tableau suivent la structure des données de l'architecture de la mémoire. En résumé, on parcourt la mémoire en faisant des sauts de byte consécutifs. Cela permet d'optimiser le nombre d'accès. À l'inverse, le mauvais programme, dont le comportement est représenté par les flèches rouges, réalise pour chaque lecture ou écriture, des sauts dans la mémoire qui sont beaucoup plus importants. Cela à pour conséquence d'augmenter significativement le nombre d'accès mémoire inutiles.

4.2 Question 1-2 : Analyse de l'outil Massif

L'outil massif de Valgrind est un outil semi-graphique de visualisation de la "heap" ou tas. Il permet de facilement se rendre compte des potentiels fuites de mémoire. Afin d'illustrer cet outil, nous avons écrit deux petits programmes un bon et un mauvais. Le bon programme est illustré dans le listing 6.

```
1 void allocation(unsigned int i)
2 {
3     cout << "Allocation de " << N << " octets" << endl;
4     tableau[i] = (unsigned char *)malloc(N);
5 }
6
7 void liberation(unsigned int i)
8 {
9     cout << "Liberation de " << N << " octets" << endl;
10    if (tableau[i])
11        free((void *)tableau[i]);
12    tableau[i] = NULL;
13 }
14
15 int main()
16 {
17     cout << "Laboratoire 4 de SeS sur massif" << endl;
18     cout << "Programme bon :)" << endl;
19     // on alloue tout le tableau
20     for (int i = 0; i < N_TAB; i++)
21     {
22         usleep(TIME_SLEEP);
23         allocation(i);
24     }
25     cout << "Tout alloue" << endl;
26     //on libere tout le tableau
27     for (int i = 0; i < N_TAB; i++)
28     {
29         usleep(TIME_SLEEP);
30         liberation(i);
31     }
32     cout << "FIN avec liberation" << endl;
33     return 0;
34 }
```

Listing 6 – Exemple d'un bon programme d'allocation dynamique

Dans l'exemple du listing 6, nous faisons une allocation dynamique pour chaque case du tableau puis nous libérons toutes ces cases. Nous devrions donc normalement allouer une certaine partie de la mémoire, puis la libérer pour ainsi revenir au niveau où elle était avant le démarrage du programme. L'outil Massif de Valgrind permet d'illustrer graphiquement ce comportement et permet donc de contrôler les source de fuite de mémoire.

Comme souhaité, nous pouvons constater sur la figure 10 que le graphique qui représente l'utilisation de la mémoire monte pour attendre un pic à 169.8KB puis redescend pour revenir au même niveau qu'au début du programme. Nous constatons donc qu'il n'y a pas eu un nombre allocations de mémoire supérieur au nombre de libérations.

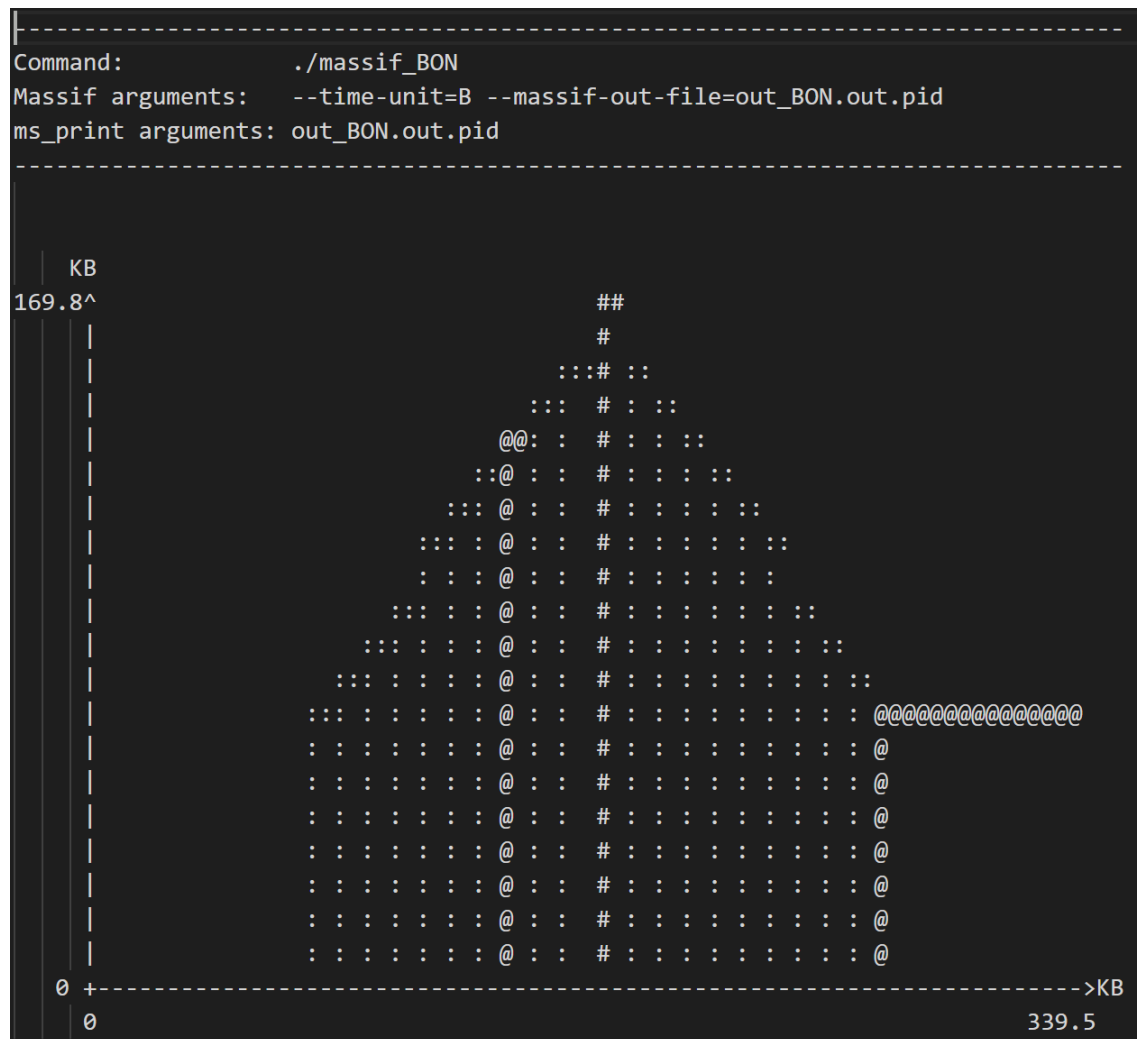


FIGURE 10 – Capture d’écran du log de sortie de massif avec un bon programme

Maintenant, reproduisons le même test, mais avec un mauvais programme. Pour l'exemple donné dans le listing 7, nous avons gardé le même principe sauf que cette fois-ci, nous n'avons fait qu'une seule libération de mémoire.

```
1 void allocation(unsigned int i)
2 {
3     cout << "Allocation de " << N << " octets" << endl;
4     tableau[i] = (unsigned char *)malloc(N);
5 }
6
7 void liberation(unsigned int i)
8 {
9     cout << "Liberation de " << N << " octets" << endl;
10    if (tableau[i])
11        free((void *)tableau[i]);
12    tableau[i] = NULL;
13 }
14
15 int main()
16 {
17     cout << "Laboratoire 4 de SeS sur massif" << endl;
18     cout << "Programme mauvais :(" << endl;
19     // on alloue tout le tableau
20     for (int i = 0; i < N_TAB; i++)
21     {
22         usleep(100000);
23         allocation(i);
24     }
25     cout << "Tout alloue" << endl;
26     //on le libere pas
27     liberation(0);
28     cout << "FIN sans liberation" << endl;
29     return 0;
30 }
```

Listing 7 – Exemple d'un mauvais programme d'allocation dynamique

La figure 11 illustre le résultat obtenu avec le mauvais programme et l'outil Massif de Valgind.

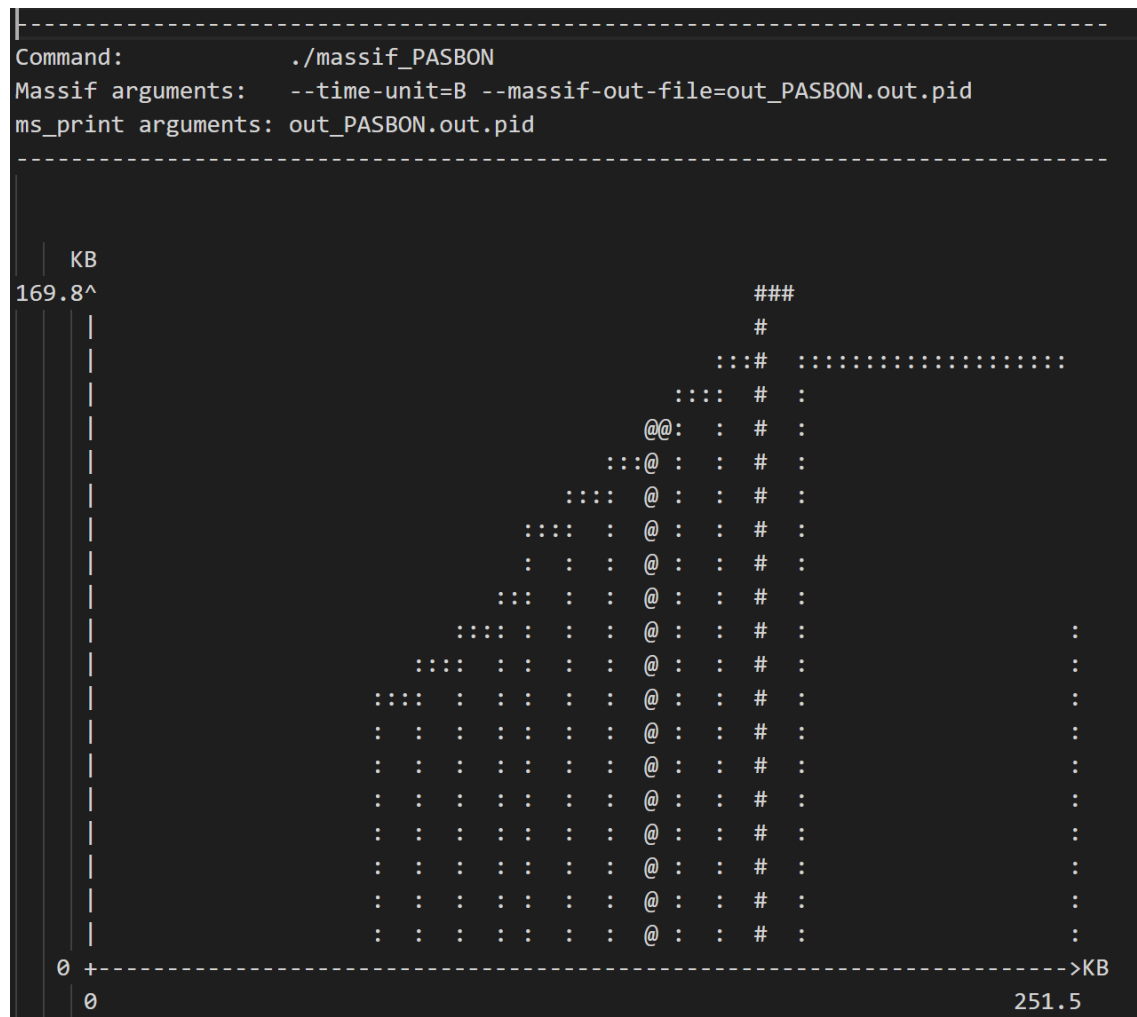


FIGURE 11 – Capture d'écran du log de sortie de massif avec un mauvais programme

Comme précédemment, nous pouvons observer sur la figure 11 que le graphique monte pour attendre un pic à 169.8KB. Cependant, une fois le pic attend, le graphique redescend juste d'un cran, puis reste au même niveau jusqu'à la fin du programme ou le MMU va finalement faire la libération. Nous avons donc bien eue une fuite de mémoire lors de l'exécution du mauvais programme. Cet outil est donc très pratique pour contrôler que toutes les allocations de mémoire sont bien libéré avant la fin du programme et qu'il n'y a pas d'éventuelle fuite de mémoire.

4.3 Question 1-3 : Analyse de l'outil Memcheck

Memcheck est un outil de Valgrind qui sert à détecter les erreurs mémoires. Cet outil est un peu plus généraliste, car il trouve aussi bien les accès mémoire trop nombreux, comme cachegrind, que les memory leak comme massif.

Voici la liste de tous les tests à suivre dans cette section.

- Accès mémoire
- Utilisation des variables non initialisées
- Libérations de la mémoire incorrect
- La mauvaise utilisation de *memcpy*
- Les fuites de mémoires

4.3.1 Essai d'accès mémoire depuis une autre pile

Le premier test que nous allons effectuer sur l'outil memcheck est un accès mémoire depuis un autre stack. Cela signifie que nous allons essayer de lire une variable qui n'appartient pas à la fonction en cours d'utilisation. Le listing 8 nous montre le code utilisé pour illustrer ce cas de figure.

```
1  int* Stack0(){
2      int varStack0 = 8;
3      return &varStack0;
4  }
5  int main() {
6
7      // Getting address to memory from another stack
8      int * pStack0 = Stack0();
9
10     printf("Essai d'accès memoire depuis une autre pile : \n
11     ");
12     printf("Valeur attendue : 8 / lue : %d\n",* pStack0);
13     printf("\n*****\n");
14     return 0;
15 }
```

Listing 8 – Exemple d'un programme de tentative d'accès à une autre pile

Comme attendu memcheck nous averti d'un accès mémoire invalide. En nous précisant qu'il s'agit d'une adresse non disponible dans le stack actuel. Le message complet est visible sur la figure 12

```
==10900== Invalid read of size 4
==10900==    at 0x40116C: main (in /home/lmi/SeS/Labo04_Valgrind/memcheck/memcheck_acces)
==10900== Address 0x0 is not stack'd, malloc'd or (recently) free'd
```

FIGURE 12 – Log de sortie de memcheck / accès mémoire

4.3.2 Essai d'utilisation d'une variable non initialisée

Le deuxième test de l'outil memcheck est la détection de l'utilisation d'une variable non initialisée. Cet outil est certes très pratique, car il est inutile lire le contenu d'une variable pas encore définie, mais ce style de problème est aussi détecté par le compilateur g++ qui ne génère pas une erreur, mais affiche tout de même un avertissement. Comme nous pouvons l'observer sur la figure 13.

```
memcheck/memcheck_2.UndefUse.c: In function 'main':
memcheck/memcheck_2.UndefUse.c:14:5: warning: 'noInit' is used uninitialized in this function [-Wuninitialized]
14 |     printf("Valeur de la variable : %d\n", noInit);
    |     ^~~~~~
```

FIGURE 13 – Log de sortie de g++ / variable non initialisée

Sur le listing 9, se trouve le code utilisé pour créer ce cas de figure.

```
1 | int main() {
2 |
3 |     int noInit;
4 |
5 |     printf("Essai d'utilisation d'une variable non init : \n
6 | ");
7 |     printf("Valeur de la variable : %d\n", noInit);
8 |     printf("\n*****\n");
9 |     return 0;
10| }
```

Listing 9 – Exemple d'un programme d'utilisation d'une variable non initialisée

Sur le log 14 nous pouvons bien observer que memcheck détecte correctement l'utilisation d'un variable non initialisée.

```
==4570== Use of uninitialised value of size 8
==4570== at 0x48B3B4B: _itoa_word (in /usr/lib64/libc-2.33.so)
==4570== by 0x48CD95B: __vfprintf_internal (in /usr/lib64/libc-2.33.so)
==4570== by 0x48B9B7E: printf (in /usr/lib64/libc-2.33.so)
==4570== by 0x40115B: main (in /home/lmi/SeS/Labo04_Valgrind/memcheck/memcheck_undefuse)
```

FIGURE 14 – Log de sortie de memcheck / variable non initialisée

4.3.3 Essai de la double libération de la mémoire

Afin de tester le troisième outil de memcheck qui est la détection de libération de mémoire incorrect. Nous avons fait une allocation dynamique suivie de deux libérations de la mémoire. Ce code est visible sur le listing 10.

```
1 | int main() {
2 |     int * pMem = (int*) malloc(sizeof(int));
3 |
4 |     printf("Essai de la double libération de la mémoire\n");
5 |     if(pMem != NULL){
6 |         free((void*)pMem);
7 |         free((void*)pMem);
8 |     }
9 |     else{
10 |         printf("Pas de mémoire à libérer !\n");
11 |     }
12 |     printf("\n*****\n");
13 |
14 |     return 0;
15 | }
```

Listing 10 – Exemple d'un programme de double libération de la mémoire

Sur le log de la figure 15, memcheck détecte bien que la deuxième libération de la mémoire n'est pas correct.

```
==4571== Invalid free() / delete / delete[] / realloc()
==4571==    at 0x48430E4: free (vg_replace_malloc.c:872)
==4571==    by 0x401184: main (in /home/lmi/SeS/Labo04_Valgrind/memcheck/memcheck_incfree)
==4571== Address 0x4a33040 is 0 bytes inside a block of size 4 free'd
==4571==    at 0x48430E4: free (vg_replace_malloc.c:872)
==4571==    by 0x401178: main (in /home/lmi/SeS/Labo04_Valgrind/memcheck/memcheck_incfree)
==4571== Block was alloc'd at
==4571==    at 0x484086F: malloc (vg_replace_malloc.c:381)
==4571==    by 0x401157: main (in /home/lmi/SeS/Labo04_Valgrind/memcheck/memcheck_incfree)
```

FIGURE 15 – Log de sortie de memcheck / libération de la mémoire

4.3.4 Essai de copie avec "memcpy"

Ce code représente la mauvaise utilisation d'un pointeur. Dans un premier temps, on alloue dynamiquement le pointeur de source puis on décale le pointeur de destination de deux cases. Comme cela, lors de la copie, on se situera dans une plage non allouée à notre programme et non initialisée.

```

1  int main() {
2      int * pT0 = (int*) malloc(sizeof(int)+2);
3      int * pFROM = (int*) ((char*) pT0 + 2);
4      *pFROM = 6;
5      printf("Essai de copie avec memcpy de trop grande valeur
6      ");
7      printf("Copie des data depuis pFROM a pT0 avec 2 bytes
8      supplémentaire\n");
9      if(pFROM != NULL){
10         printf("pFROM data / attendue = 6          / lue = %d\n",*
11         pFROM);
12         printf("pT0 data    / attendue = 393216 / lue = %d\n",*
13         pT0);
14
15         memcpy(pT0,pFROM,sizeof(int));
16         printf("pFROM data / attendue = 6          / lue = %d\n",*
17         pFROM);
18         printf("pT0 data    / attendue = 6          / lue = %d\n",*
19         pT0);
20     }
21     else{
22         printf("pFROM pas init !\n");
23     }
24     printf("\n*****\n");
25     return 0;
26 }
```

Listing 11 – Exemple d'un programme de mauvaise utilisation de memcheck

Memcheck détecte bien le saut (jump) ainsi que la partie non initialisée du pointeur de source.

```

==4572== Conditional jump or move depends on uninitialised value(s)
==4572== at 0x48CDB93: __vfprintf_internal (in /usr/lib64/libc-2.33.so)
==4572== by 0x48B9B7E: printf (in /usr/lib64/libc-2.33.so)
==4572== by 0x4011BF: main (in /home/lmi/SeS/Labo04_Valgrind/memcheck/memcheck_ovrlp)
==4572==
==4572== Use of uninitialised value of size 8
==4572== at 0x48B3B4B: _itoa_word (in /usr/lib64/libc-2.33.so)
==4572== by 0x48CD95B: __vfprintf_internal (in /usr/lib64/libc-2.33.so)
==4572== by 0x48B9B7E: printf (in /usr/lib64/libc-2.33.so)
==4572== by 0x4011BF: main (in /home/lmi/SeS/Labo04_Valgrind/memcheck/memcheck_ovrlp)
==4572==
==4572== Conditional jump or move depends on uninitialised value(s)
==4572== at 0x48B3B5C: _itoa_word (in /usr/lib64/libc-2.33.so)
==4572== by 0x48CD95B: __vfprintf_internal (in /usr/lib64/libc-2.33.so)
==4572== by 0x48B9B7E: printf (in /usr/lib64/libc-2.33.so)
==4572== by 0x4011BF: main (in /home/lmi/SeS/Labo04_Valgrind/memcheck/memcheck_ovrlp)
```

FIGURE 16 – Log de sortie de memcheck / problème avec memcpy

4.3.5 Essai de l'allocation dynamique sans libération

Ce programme alloue dynamiquement de la mémoire sans la libérer à la fin de son utilisation.

```
1  #define NB_ALLOC 20
2  int main() {
3      int * pMem = (int*) malloc(sizeof(int));
4
5      printf("Essai de l'allocation dynamique sans liberation"
6      );
7      printf("Allocation dynamique sans liberation, %d fois\n"
8      , NB_ALLOC);
9      if(pMem != NULL){
10         for (int i = 0; i < NB_ALLOC; i++) {
11             pMem = (int*) malloc(sizeof(int));
12         }
13     }
14     else{
15         printf("pMem pas init\n");
16     }
17     printf("\n*****\n");
18
19     return 0;
20 }
```

Listing 12 – Exemple d'un programme ou se trouve des fuites de mémoire

Avec ce programme, memcheck nous montre le nombre de blocs qui ont fuités ainsi que le nombre de bytes correspondent.

```
==4573== HEAP SUMMARY:
==4573==      in use at exit: 84 bytes in 21 blocks
==4573==    total heap usage: 22 allocs, 1 frees, 1,108 bytes allocated
==4573==
==4573== LEAK SUMMARY:
==4573==    definitely lost: 84 bytes in 21 blocks
==4573==    indirectly lost: 0 bytes in 0 blocks
==4573==    possibly lost: 0 bytes in 0 blocks
==4573==    still reachable: 0 bytes in 0 blocks
==4573==    suppressed: 0 bytes in 0 blocks
```

FIGURE 17 – Log de sortie de memcheck / fuites de mémoire

4.4 Question 2 : Analyse d'un code avec les outils de Valgrind

Le but de cette partie du laboratoire est de mettre en pratique les différents outils de Valgrind sur un programme complet comportant des erreurs afin de les corriger. Pour ce faire, nous avons tout d'abord compilé le projet avec g++. Ce compilateur ne nous a sorti aucun avertissement ni erreur. Puis nous avons commencé par lancer tous les outils disponibles avec Valgrind. Mais comme dit précédemment dans ce rapport, l'outil le plus pratique et celui qui regroupe toutes les informations est **memcheck**. Voici donc sur la figure 18, le log de sortie initiale du fichier *bitmap.c*.

```
==4583== Memcheck, a memory error detector
==4583== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4583== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==4583== Command: ./bitmap
==4583== Parent PID: 4566
==4583==
==4583== Invalid read of size 1
==4583==   at 0x4014BF: steganographyEncrypt(char const*, char const*, char const*) (in /home/lmi/SeS/
==4583==   by 0x40119D: main (in /home/lmi/SeS/Labo04_Valgrind/bitmap/bitmap)
==4583==   Address 0x4ec7a4e is 2 bytes after a block of size 1,132 alloc'd
==4583==   at 0x484222F: operator new[](unsigned long) (vg_replace_malloc.c:640)
==4583==   by 0x401413: steganographyEncrypt(char const*, char const*, char const*) (in /home/lmi/SeS/
==4583==   by 0x40119D: main (in /home/lmi/SeS/Labo04_Valgrind/bitmap/bitmap)
==4583==
==4583==
==4583== HEAP SUMMARY:
==4583==   in use at exit: 4,200 bytes in 1 blocks
==4583==   total heap usage: 543 allocs, 542 frees, 1,104,996 bytes allocated
==4583==
==4583== LEAK SUMMARY:
==4583==   definitely lost: 4,200 bytes in 1 blocks
==4583==   indirectly lost: 0 bytes in 0 blocks
==4583==   possibly lost: 0 bytes in 0 blocks
==4583==   still reachable: 0 bytes in 0 blocks
==4583==   suppressed: 0 bytes in 0 blocks
==4583== Rerun with --leak-check=full to see details of leaked memory
==4583==
==4583== For lists of detected and suppressed errors, rerun with: -s
==4583== ERROR SUMMARY: 240 errors from 1 contexts (suppressed: 0 from 0)
```

FIGURE 18 – Log initiale de memcheck sur le fichier *bitmap.c*

Sur cette figure, nous pouvons observer deux erreurs. La première est une lecture invalide dans la fonction **steganographyEncrypt()**. Et la seconde une fuite de mémoire d'un bloc de 4'200 bytes. La deuxième erreur n'ayant pas d'indication plus précise, nous avons décidé de commencer par l'accès invalide en lecture.

En parcourant la fonction **steganographyEncrypt()**, quelque chose nous a interpellé. La condition limite d'une boucle `for` qui était `j < (widthLoop+2)`. Ce code n'est pas du tout conventionnel, car il est très peu probable d'aller à une taille + 2, mais plutôt jusqu'à la taille. Comme le log de memcheck nous avait averti de la lecture d'un bloc, nous avons changé cette condition limite pour `j < (widthLoop+1)`. Le code modifié se trouve sur le listing 13.

```
1  ...
2  for (i=heightLoop; i; i--, bitmap.p_row++)
3  {
4      fread (hiddenText.p_buffer, hiddenText.
5      nb_byte_line, 1, pFileHiddenSource);
6      hiddenText.p_rgb = (RGB*)hiddenText.p_buffer;
7
8      unsigned short j;
9      for (j=0; j<(widthLoop+1) /*(widthLoop+2)*/; j=j+1,
10      hiddenText.p_rgb++)
11      {
12          if ((hiddenText.p_rgb->Blue != WHITE) ||
13              (hiddenText.p_rgb->Green != WHITE) ||
14              (hiddenText.p_rgb->Red != WHITE))
15          {
16              modifyPixel ((*bitmap.p_row) + j);
17          }
18      }
19  }
```

Listing 13 – Première correction d'erreur du fichier bitmap.c

Après cette première modification, nous avons relancé memcheck pour vérifier si nous avons bien trouvé l'erreur ou alors mal compris l'utilité de cette partie du code. Mais comme nous nous y attendions, l'erreur de memcheck sur une lecture invalide a disparu. Le nouveau log de memcheck est visible sur la figure 19

```
==6485== Memcheck, a memory error detector
==6485== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6485== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==6485== Command: ./bitmap_corr
==6485== Parent PID: 6463
==6485==
==6485==
==6485== HEAP SUMMARY:
==6485==   in use at exit: 4,200 bytes in 1 blocks
==6485==   total heap usage: 543 allocs, 542 frees, 1,104,996 bytes allocated
==6485==
==6485== LEAK SUMMARY:
==6485==   definitely lost: 4,200 bytes in 1 blocks
==6485==   indirectly lost: 0 bytes in 0 blocks
==6485==   possibly lost: 0 bytes in 0 blocks
==6485==   still reachable: 0 bytes in 0 blocks
==6485==   suppressed: 0 bytes in 0 blocks
==6485== Rerun with --leak-check=full to see details of leaked memory
==6485==
==6485== For lists of detected and suppressed errors, rerun with: -s
==6485== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

FIGURE 19 – Log intermédiaire de memcheck sur le fichier bitmap_corr.c

Il nous restait donc à trouver une erreur sur la libération de la mémoire allouée dynamiquement. Pour ce faire, nous avons utilisé une manière assez simple qui consiste à compter le nombre de `new` et le nombre de `delete`. Puis une fois que nous avons bien constaté le manque d'un `delete` quelque part dans le code nous avons cherché plus précisément où l'erreur se trouvait. Une fois de plus dans la fonction `steganographyEncrypt()`, où 3 variables étaient allouées dynamiquement, mais dont seulement 2 ont été libérées. Nous avons finalement trouvée laquelle n'était pas libérée puis, nous avons ajouté la ligne visible dans le listing 14.

```
1 | delete[] bitmap.p_buffer; // #corr Fuite de memoire trouver  
   | via memcheck
```

Listing 14 – Deuxième correction du fichier `bitmap.c`

Une fois cette deuxième modifications faites, nous avons relancé une fois de plus `memcheck` pour obtenir le log visible sur la figure 20 qui nous montre que plus aucune erreur n'est détectée.

```
==6619== Memcheck, a memory error detector  
==6619== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.  
==6619== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info  
==6619== Command: ./bitmap_corr  
==6619== Parent PID: 6601  
==6619==  
==6619==  
==6619== HEAP SUMMARY:  
==6619==    in use at exit: 0 bytes in 0 blocks  
==6619== total heap usage: 543 allocs, 543 frees, 1,104,996 bytes allocated  
==6619==  
==6619== All heap blocks were freed -- no leaks are possible  
==6619==  
==6619== For lists of detected and suppressed errors, rerun with: -s  
==6619== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

FIGURE 20 – Log final de `memcheck` sur le fichier `bitmap_corr.c`

Nous avons donc pu constater l'intérêt des outils de Valgrind dans la détection d'erreur et la correction d'erreur de programmation. Le code complet et corrigé de `bitmap_corr.c` se trouve dans les annexes du rapport.

5 Conclusion

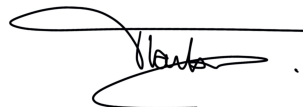
Cette série de laboratoires à permis de mettre en pratiques trois sujets différents liés au système d'exploitation Linux. La première partie sur Uboot, a permis de mettre en place la configuration de uboot ainsi que quelques modifications comme le changement du type de partition de boot, l'initialisation réseaux puis finalement une petite expérimentation de l'option de protection du stack du compilateur gcc. Cette première partie à aussi permis de se familiariser avec l'environnement. La deuxième partie était composée de beaucoup de configuration et de modification afin d'expérimenter la consolidation de la sécurité du Kernel Linux. Dans cette partie nous avons constaté qu'il était possible d'activer plusieurs options de sécurité disponible dans l'interface semi-graphique de configuration de Linux. Nous avons aussi pu observer les possibilités de modification du noyau afin d'en diminuer son volume. Il est possible de désactiver des fonctionnalités ou drivers non utilisées afin d'économiser de la place. Nous avons aussi pu observer différentes options de sécurité lors du démarrage du Kernel. Enfin la troisième partie du laboratoire était plus orienté sur la pratique et sur la programmation. Nous avons pu découvrir et mettre en pratique différents outils de Valgrind permettant de contrôler plusieurs paramètres important d'un code. Comme le contrôle du nombre d'accès mémoire, les potentielles fuites de mémoire cela permettant d'améliorer la fiabilité et l'optimisation d'un code. Ces outils sont donc utiles pour réaliser de la correction de problèmes, du contrôle fiabilité, du contrôle de performance et de l'optimisation.

6 Signatures

Lausanne le 8 avril 2022



(a) Quentin Müller



(b) Tristan Traiber

Table des figures

1	Prompt de uboot du nanoPI	3
2	Résultat de la commande <i>ifconfig</i> sur la nanoPI	6
3	Code C du test du stack protector	7
4	Test de la détection du buffer overflow	8
5	Différence entre activation <M> et [*]	9
6	Configuration à respecter pour le répertoire <i>File Systems</i>	11
7	Capture d'écran du log de sortie de cachegrind avec un bon programme	14
8	Capture d'écran du log de sortie de cachegrind avec un mauvais programme	15
9	Représentation des données en mémoire	16
10	Capture d'écran du log de sortie de massif avec un bon programme .	18
11	Capture d'écran du log de sortie de massif avec un mauvais programme	20
12	Log de sortie de memcheck / accès mémoire	21
13	Log de sortie de g++ / variable non initialisée	22
14	Log de sortie de memcheck / variable non initialisée	22
15	Log de sortie de memcheck / libération de la mémoire	23
16	Log de sortie de memcheck / problème avec memcpy	24
17	Log de sortie de memcheck / fuites de mémoire	25
18	Log initiale de memcheck sur le fichier bitmap.c	26
19	Log intermédiaire de memcheck sur le fichier bitmap_corr.c	27
20	Log final de memcheck sur le fichier bitmap_corr.c	28

7 Annexes

A Codes

A.1 Fichier *bitmap_corr.c*

```

1 //gcc -Wall -g -o bitmap bitmap.C -lstdc++
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #include <stdint.h>
6 #define WHITE 0xFF
7 #define BLACK 0x0
8 #define DELTA 100
9
10 typedef struct
11 {
12     uint16_t ImageFileType ; // Always 4D42h(''BM'')
13     uint32_t FileSize ;      // Physical file size in bytes
14     uint16_t Reserved1 ;
15     uint16_t Reserved2 ;
16     uint32_t ImageDataOffset ; // Start of image data offset
17                                // in byte
18 }__attribute__((__packed__)) BITMAP_HEADER1 ;
19
20 typedef struct
21 {
22     uint16_t HeaderSize ;      // Size of this header
23
24     uint16_t Reserved1;
25     uint16_t ImageWidth ;      // Image width in pixel
26
27     uint16_t Reserved2;
28     uint16_t ImageHeight ;     // Image height in pixel
29
30     uint16_t Reserved3;
31     uint16_t NumberOfImagePlanes ; // always 1
32     uint16_t BitPerPixel ;         // 1, 4, 8, 24
33     uint32_t CompressionMethod ;   // 0, 1, 2
34     uint32_t SizeOfBitmap ;        // Size of the bitmap in
35                                     // bytes
36     uint32_t HorzResolution ;      // Horiz. Resolution in
37                                     // pixel per meter
38     uint32_t VertResolution ;      // Vert. Resolution in
39                                     // pixel per meter
40     uint32_t NumColorsUsed ;        // Number of the colors in
41                                     // the bitmap
42     uint32_t NumSignificantColors ; // Number of important
43                                     // colors in palette
44 }__attribute__((__packed__)) BITMAP_HEADER2 ;
45
46 typedef struct
47 {
48     uint8_t Blue ;
49     uint8_t Green ;
50     uint8_t Red ;
51 }__attribute__((__packed__)) RGB ;

```

```
52 typedef struct
53 {
54     BITMAP_HEADER1  h1;
55     BITMAP_HEADER2  h2;
56     uint32_t        nb_byte_line;
57     RGB             *p_buffer;
58     RGB             *p_rgb;
59     FILE            *pFile;
60 } IMAGE_BMP;
61
62 static void steganographyEncrypt(const char *pSrc, const
        char *pDst, const char *pHiddenSource);
63
64 static void modifyPixel(RGB *p);
65
66 static void steganographyDecrypt(const char *pOri, const
        char *pModified, const char *pFoundHiddenText);
67
68 int main(void)
69 {
70     steganographyEncrypt("img1.bmp", "img2.bmp", "
        hidden_text.bmp");
71     steganographyDecrypt("img1.bmp", "img2.bmp", "
        hidden_text2.bmp");
72     return 0;
73 }
74
75 static void steganographyEncrypt(const char *pSrc, const
        char *pDst, const char *pHiddenSource)
76 {
77     FILE *pFileSource;
78     FILE *pFileDest;
79     FILE *pFileHiddenSource;
80     int i;
81
82     struct {
83         BITMAP_HEADER1  h1;
84         BITMAP_HEADER2  h2;
85         unsigned long    nb_byte_line;
86         RGB             **p_buffer;
87         RGB             **p_row;
88     } bitmap;
89
90     struct {
91         BITMAP_HEADER1  h1;
92         BITMAP_HEADER2  h2;
93         unsigned long    nb_byte_line;
94         RGB             *p_buffer;
95         RGB             *p_rgb;
96     } hiddenText;
97
98     // Open the files
99     pFileSource = fopen(pSrc, "rb");
100    pFileDest    = fopen(pDst, "wb");
101
102
```



```
103 // Read the two headers
104 fread(&bitmap.h1, sizeof(bitmap.h1), 1, pFileSource);
105 fwrite(&bitmap.h1, sizeof(bitmap.h1), 1, pFileDest);
106 fread(&bitmap.h2, sizeof(bitmap.h2), 1, pFileSource);
107 fwrite(&bitmap.h2, sizeof(bitmap.h2), 1, pFileDest);
108
109 // Reserve dynamiquement le 1er tableau de pointeurs
110 bitmap.p_buffer = (RGB**)new unsigned char[bitmap.h2.
    ImageHeight*sizeof(RGB*)];
111 memset(bitmap.p_buffer, 0, bitmap.h2.ImageHeight * sizeof
    (RGB*));
112
113 // Reserve dynamiquement toutes les lignes
114 bitmap.nb_byte_line =(bitmap.h2.ImageWidth * sizeof(RGB))
    + (bitmap.h2.ImageWidth % 4);
115
116 for(i=bitmap.h2.ImageHeight, bitmap.p_row = bitmap.
    p_buffer; i; i--, bitmap.p_row++)
117 {
118     *bitmap.p_row =(RGB*)new unsigned char[bitmap.
    nb_byte_line];
119     memset(*bitmap.p_row, 0, bitmap.nb_byte_line);
120 }
121
122 // Copie toute l'image en memoire
123 for(i=bitmap.h2.ImageHeight, bitmap.p_row = bitmap.
    p_buffer; i; i--, bitmap.p_row++)
124 {
125     fread(*bitmap.p_row, bitmap.nb_byte_line, 1, pFileSource);
126 }
127
128 // Traite le fichier cache
129 pFileHiddenSource = fopen(pHiddenSource, "rb");
130
131 fread(&hiddenText.h1, sizeof(hiddenText.h1), 1,
    pFileHiddenSource);
132 fread(&hiddenText.h2, sizeof(hiddenText.h2), 1,
    pFileHiddenSource);
133
134 hiddenText.nb_byte_line =(hiddenText.h2.ImageWidth *
    sizeof(RGB)) +(hiddenText.h2.ImageWidth % 4);
135
136 hiddenText.p_buffer =(RGB*) new unsigned char[hiddenText.
    nb_byte_line];
137
138 int heightLoop, widthLoop;
139
140 if(hiddenText.h2.ImageHeight <= bitmap.h2.ImageHeight)
141     heightLoop = hiddenText.h2.ImageHeight;
142 else
143     heightLoop = bitmap.h2.ImageHeight;
144
145 if(hiddenText.h2.ImageWidth <= bitmap.h2.ImageWidth)
146     widthLoop = hiddenText.h2.ImageWidth;
147 else
148     widthLoop = bitmap.h2.ImageWidth;
```

```
149
150     bitmap.p_row = bitmap.p_buffer;
151
152     for(i=heightLoop; i; i--, bitmap.p_row++)
153     {
154         fread(hiddenText.p_buffer, hiddenText.nb_byte_line, 1,
155             pFileHiddenSource);
156         hiddenText.p_rgb =(RGB*)hiddenText.p_buffer;
157
158         // #corr limite de la boucle
159         unsigned short j;
160         for(j=0; j<(widthLoop+1); j=j+1, hiddenText.p_rgb++)
161         {
162             if((hiddenText.p_rgb->Blue != WHITE) ||
163                 (hiddenText.p_rgb->Green!= WHITE) ||
164                 (hiddenText.p_rgb->Red  != WHITE))
165             {
166                 modifyPixel((*bitmap.p_row) + j);
167             }
168         }
169
170         delete [] hiddenText.p_buffer;
171         fclose(pFileHiddenSource);
172
173         // Copie l'image qui est en memoire dans un fichier
174         for(i=bitmap.h2.ImageHeight, bitmap.p_row = bitmap.
175             p_buffer; i; i--, bitmap.p_row++)
176         {
177             fwrite(*bitmap.p_row, bitmap.nb_byte_line,1, pFileDest);
178         }
179
180         // Libere toutes les memoires
181         for(i=bitmap.h2.ImageHeight, bitmap.p_row = bitmap.
182             p_buffer; i; i--, bitmap.p_row++)
183         {
184             delete [] *bitmap.p_row;
185         }
186         // #corr Fuite de memoire trouver via memcheck
187         delete[] bitmap.p_buffer;
188
189         fclose(pFileSource);
190         fclose(pFileDest);
191
192         // pOri=Original bitmap,pModified=Modified bitmap(read only)
193         // pFoundHiddenText = TextDecrypted(write only)
194         static void steganographyDecrypt(const char *pOri, const
195             char *pModified, const char *pFoundHiddenText)
196         {
197             IMAGE_BMP bitmapOri;
198             IMAGE_BMP bitmapModified;
199             IMAGE_BMP bitmapFoundHiddenText;
200
```

```
201 // Open the files
202 bitmapOri.pFile = fopen(pOri, "rb");
203 bitmapModified.pFile = fopen(pModified, "rb");
204 bitmapFoundHiddenText.pFile = fopen(pFoundHiddenText, "wb");
205
206 // Read the two headers
207 fread(&bitmapOri.h1, sizeof(bitmapOri.h1), 1, bitmapOri.pFile);
208 fread(&bitmapOri.h2, sizeof(bitmapOri.h2), 1, bitmapOri.pFile);
209 fread(&bitmapModified.h1, sizeof(bitmapModified.h1), 1,
210      bitmapModified.pFile);
211 fread(&bitmapModified.h2, sizeof(bitmapModified.h2), 1,
212      bitmapModified.pFile);
213
214 // Write the text file
215 fwrite(&bitmapOri.h1, sizeof(bitmapOri.h1), 1,
216      bitmapFoundHiddenText.pFile);
217 fwrite(&bitmapOri.h2, sizeof(bitmapOri.h2), 1,
218      bitmapFoundHiddenText.pFile);
219
220 // Initialize the nb byte per line and p_buffer
221 bitmapOri.nb_byte_line = (bitmapOri.h2.ImageWidth *
222      sizeof( RGB )) + (bitmapOri.h2.ImageWidth % 4);
223 bitmapOri.p_buffer = (RGB*) new unsigned char
224      [bitmapOri.nb_byte_line];
225 bitmapModified.nb_byte_line = (bitmapModified.h2.ImageWidth
226      * sizeof( RGB )) + (bitmapModified.h2.ImageWidth % 4);
227 bitmapModified.p_buffer = (RGB*) new unsigned char
228      [bitmapModified.nb_byte_line];
229 bitmapFoundHiddenText.nb_byte_line = bitmapOri.
230      nb_byte_line;
231
232 bitmapFoundHiddenText.p_buffer = (RGB*) new unsigned char
233      [bitmapFoundHiddenText.nb_byte_line];
234
235 unsigned short i;
236 for(i=bitmapOri.h2.ImageHeight; i; i--)
237 {
238     fread(bitmapOri.p_buffer, bitmapOri.nb_byte_line, 1,
239         bitmapOri.pFile);
240
241     fread(bitmapModified.p_buffer, bitmapModified.
242         nb_byte_line, 1, bitmapModified.pFile);
243
244     memset(bitmapFoundHiddenText.p_buffer, WHITE,
245         bitmapFoundHiddenText.nb_byte_line);
246
247     bitmapOri.p_rgb = (RGB*) bitmapOri.p_buffer;
248     bitmapModified.p_rgb = (RGB*) bitmapModified.p_buffer;
249     bitmapFoundHiddenText.p_rgb = (RGB*) bitmapFoundHiddenText
250         .p_buffer;
251 }
```

```
252     // Check the line
253     unsigned short j;
254     for(j=bitmapOri.h2.ImageWidth;j;j--,bitmapOri.p_rgb++,
        bitmapModified.p_rgb++, bitmapFoundHiddenText.p_rgb++)
255     {
256         // Control only one color(Red or Green or Blue)
257         if(bitmapOri.p_rgb->Red != bitmapModified.p_rgb->Red)
258         {
259             bitmapFoundHiddenText.p_rgb->Red    = BLACK;
260             bitmapFoundHiddenText.p_rgb->Green   = BLACK;
261             bitmapFoundHiddenText.p_rgb->Blue    = BLACK;
262         }
263     }
264     fwrite(bitmapFoundHiddenText.p_buffer,
        bitmapFoundHiddenText.nb_byte_line, 1,
        bitmapFoundHiddenText.pFile);
265 }
266
267 delete [] bitmapOri.p_buffer;
268 delete [] bitmapModified.p_buffer;
269 delete [] bitmapFoundHiddenText.p_buffer;
270
271 fclose(bitmapOri.pFile);
272 fclose(bitmapModified.pFile);
273 fclose(bitmapFoundHiddenText.pFile);
274 }
275
276 static void modifyPixel(RGB *p)
277 {
278     if(p->Blue >= DELTA)
279         p->Blue -= DELTA;
280     else
281         p->Blue += DELTA;
282
283     if(p->Green >= DELTA)
284         p->Green -= DELTA;
285     else
286         p->Green += DELTA;
287
288     if(p->Red >= DELTA)
289         p->Red -= DELTA;
290     else
291         p->Red += DELTA;
292 }
```

Listing 15 – Listing du programme *bitmap__coord.c*